学习 Go 语言





作者：

Miek Gieben

译者：

邢星

感谢：

Go 作者，Google

下朋友提供了帮助和贡献：

（按字母顺序）

Adam J. Gray，*Alex Sychev*，*Alexey Chernenkov*，*Andrea Spadaccini*，*Andrey Mirtchovski*，*Anthony Magro*，*Babu Sreekanth*，*Ben Bullock*，*Bob Cunningham*，*Brian Fallik*，*Cecil New*，*Damian Gryski*，*Dan Kortschak*，*David Otton*，*Fabian Becker*，*Filip Zaludek*，*Hadi Amiri*，*Haiping Fan*，*Jaap Akkerhuis*，*JC van Winkel*，*Jeroen Bulten*，*Jinpu Hu*，*John Shahid*，*Jonathan Kans*，*Joshua Stein*，*Makoto Inoue*，*Mayuresh Kathe*，*"mem"*，*Michael Stapelberg*，*Olexandr Shalakhin*，*Paulo Pinto*，*Peter Kleiweg*，*Philipp Schmidt*，*Robert Johnson*，*Russel Winder*，*Sonia Keys*，*Stefan Schroeder*，*Thomas Kapplet*，*T.J. Yang*，*"Cobold"*，*"Simoc"*，*"Uriel"*†，*邢星*。还有来自以下朋友的小帮助:

Alexander Katasonov, Daniele Pala, Iaroslav Tymchenko, Nicolas Kaiser, Marco Ynema.

“学习 Go 语言” 已经被翻译为：

- 中文，邢星；

Learning as we Go (*1.0*)

支持 Go 1.1 版本

Contents

读者 iv

1 简介 **1**

官方文档 1

Hello World 2

编译和运行代码 3

本书使用的设置 3

变量、类型和关键字 3

运算符和内建函数 7

Go 关键字 8

控制结构 8

内建函数 13

array、slices 和 map 14

练习 19

答案 21

2 函数 **26**

作用域 27

多值返回 28

命名返回值 28

延迟代码 29

变参 30

函数作为值 31

回调 31

恐慌（Panic）和恢复（Recover） 32

练习 33

答案 37

3 包 **44**

标识符 45

包的文档 46

测试包 47

常用的包 48

练习 50

答案 51

4 进阶 **54**

内存分配 54

定义自己的类型 57

转换 59

组合 60

练习 61

答案 63

5 接口 **70**
方法 72
接口名字 73
简短的例子 73
练习 78
答案 79

6 并发 **82**
更多关于 channel 84
练习 85
答案 87

7 通讯 **90**
io.Reader 91
一些例子 91
命令行参数 92
执行命令 92
网络 93
练习 94
答案 97

A 版权 **106**
贡献者 106
许可证和版权 107

B 索引 **108**

C Bibliography **110**

List of Exercises

1 (0) For-loop 19
2 (0) FizzBuzz 19
3 (1) 字符串 19
4 (1) 平均值 19
5 (0) 平均值 33
6 (0) 整数顺序 33
7 (1) 作用域 33
8 (1) 栈 33
9 (1) 变参 34
10 (1) 斐波那契 34
11 (1) map 函数 34
12 (0) 最小值和最大值 34
13 (1) 冒泡排序 34
14 (1) 函数返回一个函数 35
15 (0) stack 包 50
16 (2) 计算器 50

17 (1) 指针运算 61
18 (2) 使用 interface 的 map 函数 61
19 (1) 指针 61
20 (1) Linked List 61
21 (1) Cat 61
22 (2) 方法调用 62
23 (1) 接口和编译 78
24 (1) 指针和反射 78
25 (2) 接口和 max() 78
26 (1) Channel 85
27 (2) 斐波那契 II 85
28 (2) 进程 94
29 (0) 单词和字母统计 95
30 (0) Uniq 95
31 (2) Quine 95
32 (1) Echo 服务 95
33 (2) 数字游戏 95
34 (1) *Finger 守护进程 96

前言

> "Go 是面向对象的语言吗？是也不是。"
>
> *FAQ*
> GO AUTHORS

读者

这是关于来自 Google 的 Go 语言的简介。目标是为这个新的、革命性的语言提供一个指南。

本书假设你已经在系统中安装了 Go。

这本书的目标读者是那些熟悉编程，并且了解某些编程语言，例如 C[3]，C++[21]，Perl[5]，Java[15]，Erlang[4]，Scala[16]，Haskell[1]。这不是教你如何编程的书，只是教你如何使用 Go。

学习某样新东西，最佳的方式可能是通过编写程序来探索它。因此每章都包含了若干练习（和答案）来让你熟悉这个语言。练习标有编号 **Q**n，而 n 是一个数字。在练习编号后面的圆括号中的数字指定了该题的难度。难度范围从 0 到 2：

0. 简单；
1. 中等；
2. 困难。

其后为了容易索引，提供了一个简短的标题。例如：

Q1. (1) map 函数 …

展示了难度等级 1、编号 **Q1** 的关于 `map()` 函数的问题。相关答案在练习的下一页。答案的顺序和练习一致，以 **A**n 开头的答案，对应编号 n 的练习。一些练习没有答案，它们将用星号标记出来。

内容布局

第 1 章：简介
　讨论了语言中可用的基本类型、变量和控制结构。

第 2 章：函数
　会了解到函数，这是 Go 程序中的基本部件。

第 3 章：包
　会了解在包中整合函数和数据。同时也将了解如何对包编写文档和进行测试。

第 4 章：进阶
　会看到如何创建自定义的类型。同时也将了解 Go 中的内存分配。

第 5 章：接口
　Go 不支持传统意义上的面向对象。在 Go 中接口是核心概念。

第 6 章：并发
: 通过 go 关键字，函数可以在不同的例程（叫做 goroutines）中执行。通过 channel 来完成这些 goroutines 之间的通讯。

第 7 章：通讯
: 最后一章展示了如何用接口来完成 Go 程序的其他部分。如何创建、读取和写入文件。同时也简要了解一下网络的使用。

希望你喜欢本书，同时也喜欢上 Go 语言。

翻译

本书的内容可随意取用。这里已经有相关翻译：

- 中文，邢星：`http://www.mikespook.com/learning-go/`
- 俄文，Michael Davydenko

Miek Gieben，2011 – `miek@miek.nl`
邢星，2011 – `mikespook@gmail.com`

1 简介

> “对此感兴趣，并且希望做点什么。”
>
> *在为 Go 添加复数支持时*
> KEN THOMPSON

什么是 Go？来自其网站 [13] 的介绍：

> *Go 编程语言是一个使得程序员更加有效率的开源项目。Go 是有表达力、简洁、清晰和有效率的。它的并行机制使其很容易编写多核和网络应用，而新奇的类型系统允许构建有弹性的模块化程序。Go 编译到机器码非常快速，同时具有便利的垃圾回收和强大的运行时反射。它是快速的、静态类型编译语言，但是感觉上是动态类型的，解释型语言。*

Go 1 是 Go 语言的第一个稳定发布版本。本文档的所有练习都工作于 Go 1 – 如果不能工作，那就一定是 bug。

本书使用了下面的约定：

- 代码、关键字和注释使用 `Source Code Pro` 显示；
- 代码中的额外标识　　← *像这样显示*；
- 较长的标识提供数字 – ❶ – 详细解释在其后显示；
- （如果需要）行号在右边显示；
- shell 的例子使用 % 作为输入符；
- 用户在 shell 输入内容的例子 **用黑体显示**，系统反馈 用普通的黑体显示；
- 强调的段落会缩进，并在左边有竖线。

官方文档

Go 已经有大量的文档。例如 Go Tutorial [12] 和 Effective Go [8]。网站 `http://golang.org/doc/` 也是绝佳的起点 [a]。虽然并不一定要阅读这些文档，但是强烈建议这么做。

在互联网上搜索时，应当使用 “golang” 这个词来代替原始的 “go”。

Go 1 通过叫做 `go doc` 的标准程序提供其文档。如果你想了解内建相关（参阅下一章 “运算符和内建函数” 小节）的文档，可以像这样获取：

```
% go doc builtin
```

在第 3 章解释了如何构造你自己的包的文档。

有一些特性让 Go 与众不同。

清晰并且简洁

　　Go 努力保持小并且优美，你可以在短短几行代码里做许多事情；

[a] `http://golang.org/doc/` 本身是由 `go doc` 提供服务的。

并行
: Go 让函数很容易成为*非常*轻量的线程。这些线程在 Go 中被叫做 goroutines [b]；

Channel
: 这些 goroutines 之间的通讯由 channel[18, 25] 完成；

快速
: 编译很快，执行也很快。目标是跟 C 一样快。编译时间用秒计算；

安全
: 当转换一个类型到另一个类型的时候需要显式的转换并遵循严格的规则。Go 有垃圾收集，在 Go 中无须 `free()`，语言会处理这一切；

标准格式化
: Go 程序可以被格式化为程序员希望的（几乎）任何形式，但是官方格式是存在的。标准也非常简单：`gofmt` 的输出就是*官方认可的格式*；

类型后置
: 类型在变量名的*后面*，像这样 `var a int`，来代替 C 中的 `int a`；

UTF-8
: 任何地方都是 UTF-8 的，包括字符串以及程序代码。你可以在代码中使用 `Φ = Φ + 1`；

开源
: Go 的许可证是完全开源的，参阅 Go 发布的源码中的 LICENSE 文件；

开心
: 用 Go 写程序会非常开心！

Erlang[4] 与 Go 在部分功能上类似。Erlang 和 Go 之间主要的区别是 Erlang 是函数式语言，而 Go 是命令式的。Erlang 运行在虚拟机上，而 Go 是编译的。Go 用起来感觉更接近 Unix。

Hello World

在 Go 指南中，用一个传统的方式展现了 Go：让它打印 “Hello World”（Ken Thompson 和 Dennis Ritchie 在 20 世纪 70 年代，发布 C 语言的时候开创了这个先河）。我们不认为其他方法可以做得更好，所以就是这个吧：Go 的 “Hello World”。

Listing 1.1. Hello world

```
package main ⓿

import "fmt" ❶//实现格式化的 I/O


/* Print something */ ❷

func main() { ❸
```

[b] 是的，它的发音很接近 *co*routines，但是 goroutines 确实有一些不同，我们将在第 6 章讨论。

```
9  ❹
10         fmt.Printf("Hello, world; or καλημέρα κόσμε; or こんにちは
               世界")
11 }
```

逐行阅读这个程序。

⓿ 首行这个是必须的。所有的 Go 文件以 package `<something>`

开头，对于独立运行的执行文件必须是 package `main`；

❶ 这是说需要将 *fmt* 包加入 *main*。不是 *main* 的其他包都被称为库，其他许多编程语言有着类似的概念（参阅第 3 章）。末尾以 `//` 开头的内容是注释；

❷ 这同样是注释，不过这是被包裹于 `/*` 和 `*/` 之间的；

❸ package `main` 必须首先出现，紧跟着是 import。在 Go 中，package 总是首先出现，然后是 import，然后是其他所有内容。当 Go 程序在执行的时候，首先调用的函数是 `main.main()`，这是从 C 中继承而来。这里定义了这个函数；

❹ 第 8 行调用了来自于 *fmt* 包的函数打印字符串到屏幕。字符串由 " 包裹，并且可以包含非 ASCII 的字符。这里使用了希腊文和日文。

编译和运行代码

构建 Go 程序的最佳途径是使用 `go` 工具。

构建 `helloworld` 只需要：

```
% go build helloworld.go
```

结果是叫做 `helloworld` 的可执行文件。

```
% ./helloworld
Hello, world; or καλημέρα κόσμε; or こんにちは世界
```

本书使用的设置

- Go 被安装在 `~/go`，而 `$GOROOT` 被设置为 `GOROOT=~/go`；
- 希望编译的 Go 代码放在 `~/g/src` 而 `$GOPATH` 设置为 `GOPATH=~/g`。在使用包的时候需要用到这个变量（参阅第 3 章）。

变量、类型和关键字

在接下来的章节中，我们将会了解这个新语言的变量、基本类型、关键字和控制流。Go 在语法上有着类 C 的感觉。如果你希望将两个（或更多）语句放在一行书写，它们必须用分号 (';') 分隔。一般情况下，你不需要分号。

Go 同其他语言不同的地方在于变量的类型在变量名的*后面*。不是：int a，而是 a int。当定义了一个变量，它默认赋值为其类型的 null 值。这意味着，在 var a int后，a 的值为 0。而 var s string，意味着 s 被赋值为零长度字符串，也就是 ""。

在 Go 中，声明和赋值是两过程，但是可以连在一起。比较下面作用相同的代码片段。

Listing 1.2. 用 = 声明

```
var a int
var b bool
a = 15
b = false
```

Listing 1.3. 用:= 声明

```
a := 15
b := false
```

在左边使用了关键字 `var` 声明变量，*然后*赋值给它。右边的代码使用了 `:=` 使得在一步内完成了声明和赋值（这一形式只可用在函数内）。在这种情况下，变量的类型是由值*推演*出来的。值 15 表示是 `int` 类型，值 `false` 告诉 Go 它的类型应当是 `bool`。多个 `var` 声明可以成组；`const` 和 `import` 同样允许这么做。留意圆括号的使用：

```
var (
    x int
    b bool
)
```

有相同类型的多个变量同样可以在一行内完成声明：var x, y int 让 x 和 y 都是 `int` 类型变量。同样可以使用 平行赋值：

```
a, b := 20, 16
```

让 a 和 b 都是整数变量，并且赋值 20 给 a，16 给 b。

一个特殊的变量名是 _（下划线）。任何赋给它的值都被丢弃。在这个例子中，将 35 赋值给 b，同时丢弃 34。

```
_, b := 34, 35
```

Go 的编译器对声明却未使用的变量在报错。下面的代码会产生这个错误：声明了 `i` 却未使用

```
package main
func main() {
    var i int
}
```

布尔类型

布尔类型表示由预定义的常量 *true* 和 *false* 代表的布尔判定值。布尔类型是 `bool`。

数字类型

Go 有众所周知的类型如 int，这个类型根据你的硬件决定适当的长度。意味着在 32 位硬件上，是 32 位的；在 64 位硬件上是 64 位的。注意：int 是 32 或 64 位之一，不会定义成其他值。uint 情况相同。

如果你希望明确其长度，你可以使用 int32 或者 uint32。完整的整数类型列表（符号和无符号）是 `int8`，`int16`，`int32`，`int64` 和 `byte`，`uint8`，`uint16`，`uint32`，`uint64`。byte 是 uint8 的别名。浮点类型的值有 float32 和 float64 （没有 `float` 类型）。64 位的整数和浮点数总是 64 位的，即便是在 32 位的架构上。

需要留意的是这些类型全部都是独立的，并且混合用这些类型向变量赋值会引起编译器错误，例如下面的代码：

Listing 1.4. 相似的类型都是独立的

```
1 package main
2
3 func main() {
4         var a int       ← 通用整数类型
5         var b int32     ← 32 位整数类型
6         a = 15
7         b = a + a       ← 混合这些类型是非法的
8         b = b + 5       ← 5 是一个（未定义类型的）常量，所以这没啥问题
9 }
```

在行 7 触发一个赋值错误：

```
types.go:7: cannot use a + a (type int) as type int32 in assignment
```

赋值可以用八进制、十六进制或科学计数法：`077`，`0xFF`，`1e3` 或者 `6.022e23` 这些都是合法的。

常量

常量在 Go 中，也就是 constant。它们在编译时被创建，只能是数字、字符串或布尔值；`const x = 42` 生成 `x` 这个常量。可以使用 `iota`[c] 生成枚举值。

```
const (
        a = iota
        b = iota
)
```

第一个 `iota` 表示为 0，因此 a 等于 0，当 `iota` 再次在新的一行使用时，它的值增加了 1，因此 b 的值是 1。

也可以像下面这样，省略 Go 重复的 `= iota`：

```
const (
        a = iota
        b               ← Implicitly b = iota
)
```

如果需要，可以明确指定常量的类型：

```
const (
        a = 0           ← Is an int now
        b string = "0"
)
```

[c]单词 [iota] 在日常英语短语 “not one iota”，意思是 “不是最小”，是来自新约中的短语：“*until heaven and earth pass away, not an iota, not a dot, will pass from the Law.*”[27]

字符串

另一个重要的内建类型是 string。赋值字符串的例子：

```
s := "Hello World!"
```

字符串在 Go 中是 UTF-8 的由双引号（"）包裹的字符序列。如果你使用单引号（'）则表示一个字符（UTF-8 编码）——这种在 Go 中*不是* string。

一旦给变量赋值，字符串就不能修改了：在 Go 中字符串是不可变的。从 C 来的用户，下面的情况在 Go 中是非法的。

```
var s string = "hello"
s[0] = 'c'        ← 修改第一个字符为'c'，这会报错
```

在 Go 中实现这个，需要下面的方法：

```
s := "hello"
c := []rune(s)       ⓿
c[0] = 'c'           ❶
s2 := string(c)      ❷
fmt.Printf("%s\n", s2) ❸
```

⓿ 转换 s 为 rune 数组，查阅在第 4 章 "转换" 节、59 页的内容；

❶ 修改数组的第一个元素；

❷ 创建*新的*字符串 s2 保存修改；

❸ 用 `fmt.Printf` 函数输出字符串。

> 多行字符串
>
> 基于分号的置入（查阅文档 [8] 的 "分号" 章节），你需要小心使用多行字符串。如果这样写：
>
> ```
> s := "Starting part"
>     + "Ending part"
> ```
>
> 会被转换为：
>
> ```
> s := "Starting part";
>     + "Ending part";
> ```
>
> 这是错误的语法，应当这样写：
>
> ```
> s := "Starting part" +
>      "Ending part"
> ```
>
> Go 就不会在错误的地方插入分号。另一种方式是使用反引号 ` 作为*原始*字符串符号：
>
> ```
> s := `Starting part
>      Ending part`
> ```
>
> 留意最后一个例子 s 现在也包含换行。不像转义字符串标识，原始字符串标识的值在引号内的字符是*不*转义的。

rune

Rune 是 int32 的别名。用 UTF-8 进行编码。这个类型在什么时候使用呢？例如需要遍历字符串中的字符。可以循环每个字节（仅在使用 US ASCII 编码字符串时与字符等价，而它们在 Go 中不存在！）。因此为了获得实际的字符，需要使用 `rune` 类型。

复数

Go 原生支持复数。它的变量类型是 complex128 （64 位虚数部分）。如果需要小一些的，还有 complex64 – 32 位的虚数部分。复数写为 `re + im`i，`re` 是实数部分，`im` 是虚数部分，而 i 是标记 'i' ($\sqrt{-1}$)。使用复数的一个例子：

Printf() 的 %v 参数含义是 "用默认格式打印这个值"。

```
var c complex64 = 5+5i;fmt.Printf("Value is: %v", c)
```
将会打印：`(5+5i)`

错误

任何足够大的程序或多或少都会需要使用到错误报告。因此 Go 有为了错误而存在的内建类型，叫做 error。

`var e error` 定义了一个 error 类型的变量 `e`，其的值是 `nil`。这个 error 类型是一个接口——在第 "接口" 章将会对此进行解释。

运算符和内建函数

Go 支持普通的数字运算符，表格 1.1 列出了当前支持的运算符，以及其优先级。它们全部是从左到右结合的。

Table 1.1. 运算优先级

Precedence	Operator(s)
Highest	`* / % << >> & &^`
	`+ - | ^`
	`== != < <= > >=`
	`<-`
	`&&`
Lowest	`||`

`+ - * /` 和 `%` 会像你期望的那样工作，`& | ^` 和 `&^` 分别表示位运算符按位与，按位或，按位异或和位清除。`&&` 和 `||` 运算符是逻辑与和逻辑 或。表格中没有列出的是逻辑非：`!`。

虽然 Go 不支持运算符重载（或者方法重载），而一些内建运算符却支持重载。例如 `+` 可以用于整数、浮点数、复数和字符串（字符串相加表示串联它们）。

Go 关键字

Table 1.2. Go 中的关键字

break	default	func	interface	select
case	defer	go	map	struct
chan	else	goto	package	switch
const	fallthrough	if	range	type
continue	for	import	return	var

表格 1.2 列出了 Go 中所有的关键字。在下面的段落和章节中会介绍它们。其中有一些已经遇到过了。

- var 和 const 参阅 "变量、类型和关键字" 在第 3 页；
- 在 "Hello World" 部分，package 和 import 已经有过短暂的接触。在第 3 章对其有详细的描述。

其他都有对应的介绍和章节：

- func 用于定义函数和方法；
- return 用于从函数返回，func 和 return 参阅第 2 章了解详细信息；
- go 用于并行，参阅第 6 章；
- select 用于选择不同类型的通讯，参阅第 6 章；
- interface 参阅第 5 章；
- struct 用于抽象数据类型，参阅第 4 章；
- type 同样参阅第 4 章。

控制结构

在 Go 中只有很少的几个控制结构 [d]。例如这里没有 do 或者 while 循环，只有 for。有（灵活的）switch 语句和 if，而 switch 接受像 for 那样可选的初始化语句。还有叫做类型选择和多路通讯转接器的 select（参阅第 6 章）。语法有所不同（同 C 相比）：无需圆括号，而语句体必须总是包含在大括号内。

if-eles

在 Go 中 if 看起来是这样的：

```
if x > 0 {              ← { 是强制的
    return y
} else {
    return x
}
```

[d]这个章节复制于 [8]。

强制大括号鼓励将简单的 if 语句写在多行上。无论如何，这都是一个很好的形式，尤其是语句体中含有控制语句，例如 return 或者 break。

if 和 switch 接受初始化语句，通常用于设置一个（局部）变量。

```
if err := Chmod(0664); err != nil {      ← nil 与 C 的 NULL 类似
    fmt.Printf(err)      ← err 的作用域被限定在 if 内
    return err
}
```

可以像通常那样使用逻辑运算符（参考 1.1 表格）:

```
if true && true  {
    fmt.Println("true")
}
if ! false {
    fmt.Println("true")
}
```

在 Go 库中，你会发现当一个 if 语句不会进入下一个语句流程 – 也就是说，语句体结束于 break，continue，goto 或者 return – 不必要的 else 会被省略。

```
f, err := os.Open(name, os.O_RDONLY, 0)
if err != nil {
    return err
}
doSomething(f)
```

这个例子通常用于检测可能的错误序列。成功的流程一直执行到底部使代码很好读，当遇到错误的时候就排除它。这样错误的情况结束于 return 语句，这样就无须 else 语句。

```
f, err := os.Open(name, os.O_RDONLY, 0)
if err != nil {
    return err
}
d, err := f.Stat()
if err != nil {
    return err
}
doSomething(f, d)
```

下面的语法在 Go 中是非法的:

```
if err != nil
{                        ← 必须同 if 在同一行
    return err
}
```

参阅文档 [8] 的 “分号” 章节了解其后更深入的原因。

goto

Go 有 goto 语句——明智的使用它。用 goto 跳转到一定是当前函数内定义的标签。例如假设这样一个循环：

```
func myfunc() {
        i := 0
Here:            ← 这行的第一个词，以分号结束作为标签
        println(i)
        i++
        goto Here      ← 跳转
}
```

标签名是大小写敏感的。

for

Go 的 for 循环有三种形式，只有其中的一种使用分号。

```
for init; condition; post { }      ← 和 C 的 for 一样

for condition { }                  ← 和 while 一样

for { }                            ← 死循环
```

短声明使得在循环中声明一个序号变量更加容易。

```
sum := 0
for i := 0; i < 10; i++ {
    sum += i          ← sum = sum + i 的简化写法
}         ← i 实例在循环结束会消失
```

最后，由于 Go 没有逗号表达式，而 ++ 和 – 是语句而不是表达式，如果你想在 for 中执行多个变量，应当使用 平行赋值。

```
// Reverse a
for i, j := 0, len(a)-1; i < j; i, j = i+1, j-1 {
    a[i], a[j] = a[j], a[i]      ← 平行赋值
}
```

break 和 continue

利用 break 可以提前退出循环，break 终止当前的循环。

```
for i := 0; i < 10; i++ {
    if i > 5 {
        break      ← 终止这个循环，只打印 0 到 5
    }
    println(i)
}
```

循环嵌套循环时，可以在 break 后指定标签。用标签决定*哪个*循环被终止：

```
J:  for j := 0; j < 5; j++ {
        for i := 0; i < 10; i++ {
            if i > 5 {
                break J      ← 现在终止的是 j 循环，而不是 i 的那个
            }
            println(i)
        }
    }
```

利用 continue 让循环进入下一个迭代，而略过剩下的所有代码。下面循环打印了 0 到 5。

```
for i := 0; i < 10; i++ {
    if i > 5 {
        continue      ← 跳过循环中所有的代码println(i)
```

range

关键字 range 可用于循环。它可以在 slice、array、string、map 和 channel（参阅第 6 章）。range 是个迭代器，当被调用的时候，从它循环的内容中返回一个键值对。基于不同的内容，range 返回不同的东西。

当对 slice 或者 array 做循环时，range 返回序号作为键，这个序号对应的内容作为值。考虑这个代码：

```
list := []string{"a", "b", "c", "d", "e", "f"}     ⓿
for k, v := range list {        ❶
    // 对 k 和 v 做想做的事情❷
}
```

⓿ 创建一个字符串的 slice（参阅 “array、slices 和 map” 的第 14 页）。

❶ 用 range 对其进行循环。每一个迭代，range 将返回 int 类型的序号，string 类型的值，以 0 和 “a” 开始。

❷ k 的值为 0…5，而 v 在循环从 “a”…“f”。

也可以在字符串上直接使用 range。这样字符串被打散成独立的 Unicode 字符 [e] 并且起始位按照 UTF-8 解析。循环：

```
for pos, char := range "aΦx" {
    fmt.Printf("character '%c' starts at byte position %d\n", char
        , pos)
}
```

打印

[e]在 UTF-8 世界的字符有时被称作 runes。通常，当人们讨论字符时，多数是指 8 位字符。UTF-8 字符可能会有 32 位，称作 rune。在这个例子里，char 的类型是 rune。

```
character 'a' starts at byte position 0
character 'Φ' starts at byte position 1
character 'x' starts at byte position 3      ← Φ took 2 bytes
```

switch

Go 的 switch 非常灵活。表达式不必是常量或整数，执行的过程从上至下，直到找到匹配项，而如果 switch 没有表达式，它会匹配 true 。这产生一种可能——使用 switch 编写 if-else-if-else 判断序列。

```
func unhex(c byte) byte {
    switch {
    case '0' <= c && c <= '9':
        return c - '0'
    case 'a' <= c && c <= 'f':
        return c - 'a' + 10
    case 'A' <= c && c <= 'F':
        return c - 'A' + 10
    }
    return 0
}
```

它不会匹配失败后自动向下尝试，但是可以使用 fallthrough 使其这样做。没有 fallthrough:

```
switch i {
    case 0:  // 空的 case 体
    case 1:
        f()  // 当 i == 0 时，f 不会被调用！
}
```

而这样:

```
switch i {
    case 0:  fallthrough
    case 1:
        f()  // 当 i == 0 时，f 会被调用！
}
```

用 default 可以指定当其他所有分支都不匹配的时候的行为。

```
switch i {
    case 0:
    case 1:
        f()
    default:
        g()     // 当 i 不等于 0 或 1 时调用
}
```

分支可以使用逗号分隔的列表。

```
func shouldEscape(c byte) bool {
    switch c {
    case ' ', '?', '&', '=', '#', '+':      ← , as "or"
        return true
    }
    return false
}
```

这里有一个使用两个 switch 对字节数组进行比较的例子：

```
⓿
func Compare(a, b []byte) int {
    for i := 0; i < len(a) && i < len(b); i++ {
        switch {
        case a[i] > b[i]:
            return 1
        case a[i] < b[i]:
            return -1
        }
    }
    switch { ❶
    case len(a) < len(b):
        return -1
    case len(a) > len(b):
        return 1
    }
    return 0    ❷
}
```

⓿ 比较返回两个字节数组字典数序先后的整数。

如果 a == b 返回 0，如果 a < b 返回 -1，而如果 a > b 返回 +1；

❶ 长度不同，则不相等；

❷ 字符串相等。

内建函数

预定义了少数函数，这意味着*无需*引用任何包就可以使用它们。表格 1.3 列出了所有的内建函数。[f]

Table 1.3. Go 中的预定义函数

close	new	panic	complex
delete	make	recover	real
len	append	print	imag
cap	copy	println	

[f]可以使用命令 go doc builtin 获得关于内建类型和函数的在线文档。

这些内建函数的文档记录在跟随最近的 Go 版本一起发布的伪包 *builtin* 中。

`close`
: 用于 channel 通讯。使用它来关闭 channel，参阅第 6 章了解更多。

`delete`
: 用于在 map 中删除实例。

`len` *和* `cap`
: 可用于不同的类型，`len` 用于返回字符串、slice 和数组的长度。参阅 “array、slices 和 map” 小节了解更多关于 slice、数组和函数 `cap` 的详细信息。

`new`
: 用于各种类型的内存分配。参阅 “用 new 分配内存” 的第 55 页。

`make`
: 用于内建类型（map、slice 和 channel）的内存分配。参阅 “用 make 分配内存” 的第 55 页。

`copy`
: 用于复制 slice。参阅本章的 “slice”。

`append`
: 用于追加 slice。参阅本章的 “slice”。

`panic` *和* `recover`
: 用于异常处理机制。参阅 “恐慌（Panic）和恢复（Recover）” 的第 32 页了解更多信息。

`print` *和* `println`
: 是底层打印函数，可以在不引入 *fmt* 包的情况下使用。它们主要用于调试。

`complex`、`real` *和* `imag`
: 全部用于处理 复数。有了之前给的简单的例子，不用再进一步讨论复数了。

array、slices 和 map

可以利用 array 在列表中进行多个值的排序，或者使用更加灵活的：slice。字典或哈希类型同样可以使用，在 Go 中叫做 `map`。

array

array 由 `[n]<type>` 定义，n 标示 array 的长度，而 `<type>` 标示希望存储的内容的类型。对 array 的元素赋值或索引是由方括号完成的：

```
var arr [10]int
arr[0] = 42
arr[1] = 13
fmt.Printf("The first element is %d\n", arr[0])
```

像 var `arr = [10]`int 这样的数组类型有固定的大小。大小是类型的一*部分*。由于不同的大小是不同的类型，因此不能改变大小。数组同样是值类型的：将一个数组赋值

给另一个数组，会*复制*所有的元素。尤其是当向函数内传递一个数组的时候，它会获得一个数组的副本，而不是数组的指针。

可以像这样声明一个数组：var a [3]int，如果不使用零来初始化它，则用复合声明：a := [3]int{1, 2, 3} 也可以简写为 a := [...]int{1, 2, 3}，Go 会自动统计元素的个数。

复合声明允许你直接将值赋值给 array、slice 或者 map。
参阅第 56 页的"构造函数与复合声明"了解更多信息。

注意，所有项目必须都指定。因此，如果你使用多维数组，有一些内容你必须录入：

```
a := [3][2]int{ [2]int{1,2}, [2]int{3,4}, [2]int{5,6} }
```

类似于：

```
a := [3][2]int{ [...]int{1,2}, [...]int{3,4}, [...]int{5,6} }
```

声明一个 array 时，你*必须*在方括号内输入些内容，数字或者三个点 (...)。在很久之前，这个语法被进一步简化，这里是来自之前的发布日志：

> *array、slice 和 map 的复合声明变得更加简单。使用复合声明的 array、slice 和 map，元素复合声明的类型与外部一致，则可以省略。*

这表示上面的例子可以修改为：

```
a := [3][2]int{ {1,2}, {3,4}, {5,6} }
```

slice

slice 与 array 接近，但是在新的元素加入的时候可以增加长度。slice 总是指向底层的一个 array。slice 是一个*指向* array 的指针，这是其与 array 不同的地方；slice 是引用类型，这意味着当赋值某个 slice 到另外一个变量，两个引用会指向同一个 array。例如，如果一个函数需要一个 slice 参数，在其内对 slice 元素的修改也会体现在函数调用者中，这和传递底层的 array 指针类似。通过：

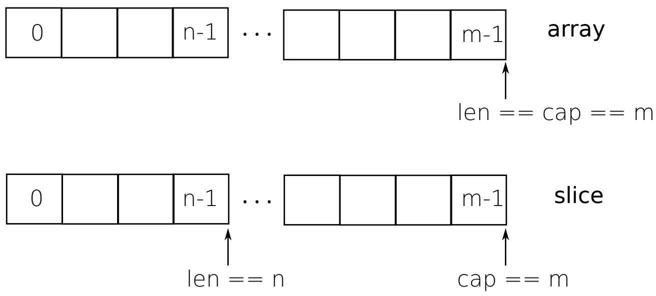
引用类型使用 make 创建。

```
sl := make([]int, 10)
```

创建了一个保存有 10 个元素的 slice。需要注意的是底层的 array 并无不同。slice 总是与一个固定长度的 array 成对出现。其影响 slice 的容量和长度。 图 1.1 描述了下面的 Go 代码。首先创建了 m 个元素长度的 array，元素类型 int：var array[m]int
然后对这个 array 创建 slice：slice := array[0:n]
然后现在有：

Figure 1.1. array 与 slice 对比

len(slice)== n ; cap(slice)== m ; len(array)== cap(array)== m .



给定一个 array 或者其他 slice，一个新 slice 通过 `a[I:J]` 的方式创建。这会创建一个新的 slice，指向变量 `a`，从序号 `I` 开始，结束在序号 `J` 之前。长度为 `J - I`。

```
// array[n:m] 从 array 创建了一个 slice，具有元素 n 到 m-1
a := [...]int{1, 2, 3, 4, 5} ⓿
s1 := a[2:4] ❶
```

```
s2 := a[1:5] ❷
s3 := a[:]   ❸
s4 := a[:4]  ❹
s5 := s2[:]  ❺
```

⓿ 定义一个 5 个元素的 array，序号从 0 到 4；

❶ 从序号 2 至 3 创建 slice，它包含元素 `3`，`4`；

❷ 从序号 1 至 4 创建，它包含元素 `2`，`3`，`4`，`5`；

❸ 用 array 中的所有元素创建 slice，这是 `a[0:len(a)]` 的简化写法；

❹ 从序号 0 至 3 创建，这是 `a[0:4]` 的简化写法，得到 `1`，`2`，`3`，`4`；

❺ 从 slice `s2` 创建 slice，注意 `s5` 仍然指向 array `a`。

在 1.5 列出的代码中，我们在第八行尝试做一些错误的事情，让一些东西超出范围（底层 array 的最大长度），然后得到了一个运行时错误。

Listing 1.5. array 和 slice

```
1 package main
2
3 func main() {
4 var array [100]int          ← Create array, index from 0 to 99
5 slice := array[0:99]        ← Create slice, index from 0 to 98
6
7 slice[98] = 'a'             ← OK
8 slice[99] = 'a'             ← Error: "throw: index out of range"
9 }
```

如果你想要扩展 slice，有一堆内建函数让你的日子更加好过一些：append 和 copy。来自于 [10]：

> *函数 append 向 slice* `s` *追加零值或其他* `x` *值，并且返回追加后的新的、与* `s` *有相同类型的 slice。如果* `s` *没有足够的容量存储追加的值，append 分配一个足够大的、新的 slice 来存放原有 slice 的元素和追加的值。因此，返回的 slice 可能指向不同的底层 array。*

```
s0 := []int{0, 0}
s1 := append(s0, 2)⓿
s2 := append(s1, 3, 5, 7)❶
s3 := append(s2, s0...)❷
```

⓿ 追加一个元素，`s1 == []int{0, 0, 2}`；

❶ 追加多个元素，`s2 == []int{0, 0, 2, 3, 5, 7}`；

❷ 追加一个 slice，`s3 == []int{0, 0, 2, 3, 5, 7, 0, 0}`。注意这三个点！

还有

> *函数 copy 从源 slice src 复制元素到目标 dst，并且返回复制的元素的个数。源和目标可能重叠。元素复制的数量是 len(src) 和 len(dst) 中的最小值。*

```
var a = [...]int{0, 1, 2, 3, 4, 5, 6, 7}
var s = make([]int, 6)
n1 := copy(s, a[0:])     ← n1 == 6, s == []int{0, 1, 2, 3, 4, 5}
n2 := copy(s, s[2:])     ← n2 == 4, s == []int{2, 3, 4, 5, 4, 5}
```

map

许多语言都内建了类似的类型，例如 Perl 有哈希，Python 有字典，而 C++ 同样也有 map（作为库）。在 Go 中有 map 类型。map 可以认为是一个用字符串做索引的数组（在其最简单的形式下）。下面定义了 map 类型，用于将 string （月的缩写）转换为 int – 那个月的天数。一般定义 map 的方法是：map[<from type>]<to type>

```
monthdays := map[string]int{
        "Jan": 31, "Feb": 28, "Mar": 31,
        "Apr": 30, "May": 31, "Jun": 30,
        "Jul": 31, "Aug": 31, "Sep": 30,
        "Oct": 31, "Nov": 30, "Dec": 31,    ← 逗号是必须的
}
```

留意，当只需要声明一个 map 的时候，使用 make 的形式：monthdays := make(map[string]int)

当在 map 中索引（搜索）时，使用方括号。例如打印出 12 月的天数：fmt.Printf("%d\n", monthdays["Dec"])

当对 array、slice、string 或者 map 循环遍历的时候，range 会帮助你，每次调用，它都会返回一个键和对应的值。

```
year := 0
for _, days := range monthdays {    ← 键没有使用，因此用 _, days
    year += days
}
fmt.Printf("Numbers of days in a year: %d\n", year)
```

向 map 增加元素，可以这样做：

```
monthdays["Undecim"] = 30    ← 添加一个月
monthdays["Feb"]     = 29    ← 闰年时重写这个元素
```

检查元素是否存在，可以使用下面的方式 [19]：

```
var value int
var present bool

value, present = monthdays["Jan"]    ← 如果存在，present 则有值 true
                                     ← 或者更接近 Go 的方式
v, ok := monthdays["Jan"]            ← “逗号 ok”形式
```

也可以从 `map` 中移除元素：

```
delete(monthdays, "Mar")
```
← *删除"Mar" 吧，总是下雨的月份*

通常来说语句 `delete(m, x)` 会删除 map 中由 `m[x]` 建立的实例。

练习

Q1. (0) For-loop

1. 创建一个基于 `for` 的简单的循环。使其循环 10 次，并且使用 *fmt* 包打印出计数器的值。
2. 用 `goto` 改写 1 的循环。关键字 `for` 不可使用。
3. 再次改写这个循环，使其遍历一个 array，并将这个 array 打印到屏幕上。

Q2. (0) FizzBuzz

1. 解决这个叫做 Fizz-Buzz[23] 的问题：

   编写一个程序，打印从 1 到 100 的数字。当是三个倍数就打印 "Fizz" 代替数字，当是五的倍数就打印 "Buzz"。当数字同时是三和五的倍数时，打印 "FizzBuzz"。

Q3. (1) 字符串

1. 建立一个 Go 程序打印下面的内容（到 100 个字符）：

```
A
AA
AAA
AAAA
AAAAA
AAAAAA
AAAAAAA
...
```

2. 建立一个程序统计字符串里的字符数量：

   `asSASA ddd dsjkdsjs dk`

   同时输出这个字符串的字节数。*提示：* 看看 *unicode/utf8* 包。
3. 扩展/修改上一个问题的程序，替换位置 4 开始的三个字符为 "abc"。
4. 编写一个 Go 程序可以逆转字符串，例如 "foobar" 被打印成 "raboof"。*提示：*不幸的是你需要知道一些关于转换的内容，参阅 "转换" 第 59 页的内容。

Q4. (1) 平均值

1. 编写计算一个类型是 `float64` 的 slice 的平均值的代码。在稍候的练习 Q5 中将会改写为函数。

答案

A1. (0) For-loop

1. 有许多种解法，其中一种可能是：

   Listing 1.6. 循环示例

   ```
   package main

   import "fmt"

   func main() {
       for i := 0; i < 10; i++ {      ← See page 10
           fmt.Printf("%d\n", i)
       }
   }
   ```

   编译并观察输出。

   ```
   % go build for.go
   % ./for
   0
   1
   .
   .
   .
   9
   ```

2. 改写的循环最终看起来像这样（仅显示了 main 函数）：

   ```
   func main() {
           i := 0          ← 定义循环变量
   Loop:                   ← 定义标签
           fmt.Printf("%d\n", i)
           if i < 10 {
               i++
               goto Loop       ← 跳转回标签
           }
   }
   ```

3. 下面是可能的解法之一：

   Listing 1.7. 用于数组的 for 循环

   ```
   func main() {
      var arr [10]int        ← Create an array with 10 elements
      for i := 0; i < 10; i++ {
          arr[i] = i        ← Fill it one by one
      }
      fmt.Printf("%v", arr)       ← With %v Go prints the value for us
   ```

```
}
```

也可以用复合声明的硬编码来实现这个：

```
a := [...]int{0,1,2,3,4,5,6,7,8,9}      ← 让 Go 来计数
fmt.Printf("%v\n", a)
```

A2. (0) FizzBuzz

1. 下面简单的程序，是一种解决办法。

Listing 1.8. Fizz-Buzz

```
package main

import "fmt"

func main() {
        const (
                FIZZ = 3 ⓿
                BUZZ = 5
        )
        var p bool      ❶
        for i := 1; i < 100; i++ { ❷;
                p = false
                if i%FIZZ == 0 { ❸
                        fmt.Printf("Fizz")
                        p = true
                }
                if i%BUZZ == 0 { ❹
                        fmt.Printf("Buzz")
                        p = true
                }
                if !p { ❺
                        fmt.Printf("%v", i)
                }
                fmt.Println() ❻
        }
}
```

⓿ 为了提高代码的可读性，定义两个常量。参阅 “常量”；

❶ 判断是否需要打印内容；

❷ for 循环，参阅 “for”

❸ 如果能被 FIZZ 整除，打印 “Fizz”；

❹ 如果能被 BUZZ 整除，打印 “Buzz”。注意，FizzBuzz 的情况已经被处理了；

❺ 如果 FIZZ 和 BUZZ 都没有打印，打印原始值；

❻ 换行。

A3. (1) 字符串

1. 这是一个解法：

Listing 1.9. 字符串

```
package main

import "fmt"

func main() {
        str := "A"
        for i := 0; i < 100; i++ {
                fmt.Printf("%s\n", str)
                str = str + "A"      ← String concatenation
        }
}
```

2. 为了解决这个问题，需要 *unicode/utf8* 包的帮助。首先，阅读一下文档 `go doc unicode/utf8 | less`。在阅读文档的时候，会注意到 func `RuneCount(p []byte)`int。然后，将 *string* 转换为 `byte` slice：

```
str := "hello"
b   := []byte(str)      ← 转换，参阅第 59 页
```

将这些整合到一起，得到下面的程序。

Listing 1.10. 字符串中的 rune

```
package main

import (
        "fmt"
        "unicode/utf8"
)

func main() {
        str := "dsjkdshdjsdh....js"
        fmt.Printf("String %s\nLength: %d, Runes: %d\n", str,
                len([]byte(str)), utf8.RuneCount([]byte(str)))
}
```

```
package main

import (
        "fmt"
)
```

3. 如下几行代码：

```
func main() {
        s := "□□□ □□□□□□□□□□□  "
        r := []rune(s)
        copy(r[4:4+3], []rune("abc"))
        fmt.Printf("Before: %s\n", s);
        fmt.Printf("After : %s\n", string(r))
}
```

4. 可以用下面的方法逆转字符串。我们从左边（i）至右（j）的交换字符，就像这样：

Listing 1.11. Reverse a string

```
import "fmt"

func main() {
  s := "foobar"
  a := []rune(s)       ← Again a conversion
  for i, j := 0, len(a)-1; i < j; i, j = i+1, j-1 {
     a[i], a[j] = a[j], a[i]      ← Parallel assignment
  }
  fmt.Printf("%s\n", string(a))      ← Convert it back
}
```

A4. (1) 平均值

1. 下面的代码计算了平均值。

```
sum := 0.0
switch len(xs) {
case 0:                 ⓿
        avg = 0
default:                ❶
        for _, v := range xs {
                sum += v
        }
        avg = sum / float64(len(xs)) ❷
}
```

⓿ 如果长度是零，返回 0；

❶ 否则计算平均值；

❷ 为了能够进行除法，必须将值转换为 `float64`。

2 函数

> "我总是兴奋于阳光的轻抚和沉寂在早期编程语言中。无需太多文字；许多已经完成了。旧的程序阅读起来就像是同表达良好的研究工作者或受到良好训练的机器同事沟通一样，而不是与编译器争论。谁愿意让其成熟到发出这样的声音呢？"
>
> RICHARD P. GABRIEL

函数是构建 Go 程序的基础部件；所遇有趣的事情都是在它其中发生的。函数的定义看起来像这样：

Listing 2.1. 函数定义

```
type mytype int       ← 新的类型，参阅第 4 章

func (p mytype) funcname(q int) (r,s int) { return 0,0 }
 ⓿       ❶          ❷        ❸       ❹          ❺
```

⓿ 关键字 func 用于定义一个函数；

❶ 函数可以绑定到特定的类型上。这叫做 *接收者*。有接收者的函数被称作 method。第 5 章将对其进行说明；

❷ *funcname* 是你函数的名字；

❸ int 类型的变量 q 作为输入参数。参数用 *pass-by-value* 方式传递，意味着它们会被复制；

❹ 变量 r 和 s 是这个函数的 命名返回值。在 Go 的函数中可以返回多个值。参阅第 28 页的 "多值返回"。如果不想对返回的参数命名，只需要提供类型：(int,int)。如果只有一个返回值，可以省略圆括号。如果函数是一个子过程，并且没有任何返回值，也可以省略这些内容；

❺ 这是函数体。注意 return 是一个语句，所以包裹参数的括号是可选的。

这里有两个例子，左边的函数没有返回值，右边的只是简单的将输入返回。

```
func subroutine(in int) { return }

func identity(in int) int { return in }
```

可以随意安排函数定义的顺序，编译器会在执行前扫描每个文件。所以函数原型在 Go 中都是过期的旧物。Go 不允许函数嵌套，然而你可以利用匿名函数实现它，参阅本章第 31 页的 "函数作为值"。

递归函数跟其他语言是一样的：

Listing 2.2. 递归函数

```
func rec(i int) {
   if i == 10 {
        return
   }
   rec(i+1)
   fmt.Printf("%d ", i)
}
```

这会打印：`9 8 7 6 5 4 3 2 1 0`。

作用域

在 Go 中，定义在函数外的变量是全局的，那些定义在函数内部的变量，对于函数来说是局部的。如果命名覆盖——一个局部变量与一个全局变量有相同的名字——在函数执行的时候，局部变量将覆盖全局变量。

Listing 2.3. 局部作用域

```
package main

var a = 6

func main() {
    p()
    q()
    p()
}

func p() {
    println(a)
}

func q() {
    a := 5        ← 定义
    println(a)
}
```

Listing 2.4. 全局作用域

```
package main

var a = 6

func main() {
    p()
    q()
    p()
}

func p() {
    println(a)
}

func q() {
    a = 5     ← 赋值
    println(a)
}
```

在 2.3 中定义了函数 `q()` 的局部变量 `a`。局部变量 `a` 仅在 `q()` 中可见。这也就是为什么代码会打印：`656`。在 2.4 中没有定义局部变量，只有全局变量 `a`。这将使得对 `a` 的赋值全局可见。这段代码将会打印：`655`。

在下面的例子中，我们在 `f()` 中调用 `g()`：

Listing 2.5. 当函数调用函数时的作用域

```
package main
```

```
var a int

func main() {
        a = 5
        println(a)
        f()
}

func f() {
        a := 6
        println(a)
        g()
}

func g() {
        println(a)
}
```

输出内容将是：565。*局部*变量*仅仅*在执行定义它的函数时有效。

多值返回

Go 一个非常特别的特性（对于编译语言而言）是函数和方法可以返回多个值（Python 和 Perl 同样也可以）。这可以用于改进一大堆在 C 程序中糟糕的惯例用法：修改参数的方式，返回一个错误（例如遇到 EOF 则返回 -1）。在 Go 中，Write 返回一个计数值和一个错误："是的，你写入了一些字节，但是由于设备异常，并不是全部都写入了。"。*os* 包中的 *File.Write 是这样声明的：

```
func (file *File) Write(b []byte) (n int, err error)
```

如同文档所述，它返回写入的字节数，并且当 n != len(b) 时，返回非 nil 的 error。这是 Go 中常见的方式。

元组没有作为原生类型出现，所以多返回值可能是最佳的选择。你可以精确的返回希望的值，而无须重载域空间到特定的错误信号上。

命名返回值

Go 函数的返回值或者结果参数可以指定一个名字，并且像原始的变量那样使用，就像输入参数那样。如果对其命名，在函数开始时，它们会用其类型的零值初始化。如果函数在不加参数的情况下执行了 return 语句，结果参数会返回。用这个特性，允许（再一次的）用较少的代码做更多的事 [a]。

名字不是强制的，但是它们可以使得代码更加健壮和清晰：*这是文档*。例如命名 int 类型的 nextPos 返回值，就能说明哪个代表哪个。

```
func nextInt(b []byte, pos int) (value, nextPos int) { /* ... */ }
```

[a] 这是 Go 的格言："用*更少*的代码做*更多*的事"。

由于命名结果会被初始化并关联于无修饰的 `return`，它们可以非常简单并且清晰。这里有一段 `io.ReadFull` 的代码，很好的运用了它：

```
func ReadFull(r Reader, buf []byte) (n int, err error) {
    for len(buf) > 0 && err == nil {
        var nr int
        nr, err = r.Read(buf)
        n += nr
        buf = buf[nr:len(buf)]
    }
    return
}
```

延迟代码

假设有一个函数，打开文件并且对其进行若干读写。在这样的函数中，经常有提前返回的地方。如果你这样做，就需要关闭正在工作的文件描述符。这经常导致产生下面的代码：

Listing 2.6. 没有 defer

```
func ReadWrite() bool {
    file.Open("file")
    // 做一些工作
    if failureX {
        file.Close()     ←
        return false
    }

    if failureY {
        file.Close()     ←
        return false
    }
    file.Close()     ←
    return true
}
```

在这里有许多重复的代码。为了解决这些，Go 有了 `defer` 语句。在 `defer` 后指定的函数会在函数退出*前*调用。

上面的代码可以被改写为下面这样。将 `Close` 对应的放置于 `Open` 后，能够使函数更加可读、健壮。

Listing 2.7. 有 defer

```
func ReadWrite() bool {
    file.Open("file")
    defer file.Close()      ← file.Close() 被添加到了 defer 列表
    // 做一些工作
    if failureX {
```

```
        return false        ← Close() 现在自动调用
    }
    if failureY {
        return false        ← 这里也是
    }
    return true             ← And here
}
```

可以将多个函数放入 "延迟列表"中，这个例子来自 [8]：

```
for i := 0; i < 5; i++ {
    defer fmt.Printf("%d ", i)
}
```

延迟的函数是按照后进先出（LIFO）的顺序执行，所以上面的代码打印：`4 3 2 1 0`。
利用 `defer` 甚至可以修改返回值，假设正在使用命名结果参数和函数符号 [b]，例如：

Listing 2.8. 函数符号

```
defer func() {
        /* ... */
}()                     ← () 在这里是必须的
```

或者这个例子，更加容易了解为什么，以及在哪里需要括号：

Listing 2.9. 带参数的函数符号

```
defer func(x int) {
        /* ... */
}(5)        ← 为输入参数 x 赋值 5
```

在这个（匿名）函数中，可以访问任何命名返回参数：

Listing 2.10. 在 defer 中访问返回值

```
func f() (ret int) {     ← ret 初始化为零
        defer func() {
                ret++     ← ret 增加为 1
        }()
        return 0     ← 返回的是 1 而不是 0！
}
```

变参

接受不定数量的参数的函数叫做变参函数。定义函数使其接受变参：

```
func myfunc(arg ...int) {}
```

`arg ...int` 告诉 Go 这个函数接受不定数量的参数。注意，这些参数的类型全部是 `int`。
在函数体中，变量 `arg` 是一个 int 类型的 slice：

[b] 函数符号也就是被叫做闭包的东西。

```
for _, n := range arg {
    fmt.Printf("And the number is: %d\n", n)
}
```

如果不指定变参的类型，默认是空的接口 `interface{}`（参阅第 5 章）。假设有另一个变参函数叫做 `myfunc2`，下面的例子演示了如何向其传递变参：

```
func myfunc(arg ...int) {
        myfunc2(arg...)      ← 按原样传递
        myfunc2(arg[:2]...)      ← 传递部分
}
```

函数作为值

就像其他在 Go 中的其他东西一样，函数也是值*而已*。它们可以像下面这样赋值给变量：

Listing 2.11. 匿名函数

```
func main() {
        a := func() {        ← 定义一个匿名函数，并且赋值给 a
          println("Hello")
        }                    ← 这里没有 ()
        a()                  ← 调用函数
}
```

如果使用 `fmt.Printf("\%T\n", a)` 打印 a 的类型，输出结果是 `func()`。

函数作为值，也会被用在其他地方，例如 map。这里将整数转换为函数：

Listing 2.12. 使用 map 的函数作为值

```
var xs = map[int]func() int{
    1: func() int { return 10 },
    2: func() int { return 20 },
        3: func() int { return 30 },      ← 必须有逗号
    /* ... */
}
```

也可以编写一个接受函数作为参数的函数，例如用于操作 `int` 类型的 slice 的 `Map` 函数。这是一个留给读者的练习，参考在第 34 页的练习 Q11。

回调

由于函数也是值，所以可以很容易的传递到其他函数里，然后可以作为回调。首先定义一个函数，对整数做一些 “事情”：

```
func printit(x int) {     ← 函数无返回值
    fmt.Printf("%v\n", x)      ← 仅仅打印
}
```

这个函数的标识是 func printit(int)，或者没有函数名的：func(int)。创建新的函数使用这个作为回调，需要用到这个标识：

```
func callback(y int, f func(int)) {     ← f 将会保存函数
    f(y)          ← 调用回调函数 f 输入变量 y
}
```

恐慌（Panic）和恢复（Recover）

Go 没有像 Java 那样的异常机制，例如你无法像在 Java 中那样抛出一个异常。作为替代，它使用了恐慌和恢复（panic-and-recover）机制。一定要记得，这应当作为最后的手段被使用，你的代码中应当没有，或者很少的令人恐慌的东西。这是个强大的工具，明智的使用它。那么，应该如何使用它呢。

下面的描述来自于 [7]：

Panic

是一个内建函数，可以中断原有的控制流程，进入一个令人恐慌的流程中。当函数 `F` 调用 `panic`，函数 `F` 的执行被中断，并且 `F` 中的延迟函数会正常执行，然后 `F` 返回到调用它的地方。在调用的地方，`F` 的行为就像调用了 `panic`。这一过程继续向上，直到程序崩溃时的所有 goroutine 返回。

恐慌可以直接调用 `panic` 产生。也可以由*运行时错误*产生，例如访问越界的数组。

Recover

是一个内建的函数，可以让进入令人恐慌的流程中的 goroutine 恢复过来。recover *仅在延迟*函数中有效。

在正常的执行过程中，调用 `recover` 会返回 `nil` 并且没有其他任何效果。如果当前的 goroutine 陷入恐慌，调用 `recover` 可以捕获到 `panic` 的输入值，并且恢复正常的执行。

这个函数检查作为其参数的函数在执行时是否会产生 panic[c]：

```
func throwsPanic(f func()) (b bool) { ⓿
defer func() { ❶
        if x := recover(); x != nil {
            b = true
        }
    }()
    f() ❷
    return ❸
}
```

⓿ 定义一个新函数 `throwsPanic` 接受一个函数作为参数（参看 “函数作为值”）。函数 `f` 产生 panic，就返回 true，否则返回 false；

[c]复制于 Eleanor McHugh 的演讲稿

❶ 定义了一个利用 `recover` 的 `defer` 函数。*如果*当前的 goroutine 产生了 panic，这个 defer 函数能够发现。当 `recover()` 返回非 `nil` 值，设置 b 为 true；

❷ 调用作为参数接收的函数。

❸ 返回 b 的值。由于 b 是命名返回值

（第 28 页），无须指定 b。

练习

Q5. (0) 平均值

1. 编写一个函数用于计算一个 `float64` 类型的 slice 的平均值。

Q6. (0) 整数顺序

1. 编写函数，返回其（两个）参数正确的（自然）数字顺序：
   `f(7,2)` → `2,7`
   `f(2,7)` → `2,7`

Q7. (1) 作用域

1. 下面的程序有什么错误？

```
package main

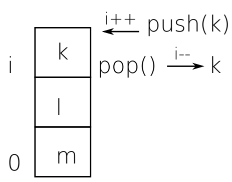
import "fmt"

func main() {
	for i := 0; i < 10; i++ {
		fmt.Printf("%v\n", i)
	}
	fmt.Printf("%v\n", i)
}
```

Q8. (1) 栈

1. 创建一个固定大小保存整数的栈。它无须超出限制的增长。定义 push 函数——将数据放入栈，和 pop 函数——从栈中取得内容。栈应当是后进先出（LIFO）的。

Figure 2.1. 一个简单的 LIFO 栈



2. 更进一步。编写一个 `String` 方法将栈转化为字符串形式的表达。可以这样的方式打印整个栈：`fmt.Printf("My stack %v\n", stack)`

   栈可以被输出成这样的形式：`[0:m] [1:l] [2:k]`

Q9. (1) 变参

1. 编写函数接受整数类型变参，并且每行打印一个数字。

Q10. (1) 斐波那契

1. 斐波那契数列以：$1, 1, 2, 3, 5, 8, 13, \ldots$ 开始。或者用数学形式表达：$x_1 = 1; x_2 = 1; x_n = x_{n-1} + x_{n-2} \quad \forall n > 2$。

   编写一个接受 `int` 值的函数，并给出这个值得到的斐波那契数列。

Q11. (1) map 函数 `map()` 函数是一个接受一个函数和一个列表作为参数的函数。函数应用于列表中的每个元素，而一个新的包含有计算结果的列表被返回。因此：

$$\text{map}(f(), (a_1, a_2, \ldots, a_{n-1}, a_n)) = (f(a_1), f(a_2), \ldots, f(a_{n-1}), f(a_n))$$

1. 编写 Go 中的简单的 `map()` 函数。它能工作于操作整数的函数就可以了。
2. 扩展代码使其工作于字符串列表。

Q12. (0) 最小值和最大值

1. 编写一个函数，找到 `int` slice (`[]int`) 中的最大值。
2. 编写一个函数，找到 `int` slice (`[]int`) 中的最小值。

Q13. (1) 冒泡排序

1. 编写一个针对 int 类型的 slice 冒泡排序的函数。这里 [24]：

   > *它在一个列表上重复步骤来排序，比较每个相邻的元素，并且顺序错误的时候，交换它们。一遍一遍扫描列表，直到没有交换为止，这意味着列表排序完成。算法得名于更小的元素就像 "泡泡" 一样冒到列表的顶端。*

   [24] 这里有一个过程代码作为示例：

```
procedure bubbleSort( A : list of sortable items )
  do
    swapped = false
    for each i in 1 to length(A) - 1 inclusive do:
      if A[i-1] > A[i] then
        swap( A[i-1], A[i] )
        swapped = true
      end if
    end for
  while swapped
end procedure
```

Q14. (1) 函数返回一个函数

1. 编写一个函数返回另一个函数，返回的函数的作用是对一个整数 $+2$。函数的名称叫做 `plusTwo`。然后可以像下面这样使用：

   ```
   p := plusTwo()
   fmt.Printf("%v\n", p(2))
   ```

   应该打印 4。参阅第 31 页的 “回调” 小节了解更多相关信息。

2. 使 1 中的函数更加通用化，创建一个 `plusX(x)` 函数，返回一个函数用于对整数加上 `x`。

答案

A5. (0) 平均值

1. 下面的函数计算平均值。

Listing 2.13. Go 中的平均值函数

```
func average(xs []float64) (avg float64) { ⓿
        sum := 0.0
        switch len(xs) {
        case 0:                 ❶
                avg = 0
        default:                ❷
                for _, v := range xs {
                        sum += v
                }
                avg = sum / float64(len(xs)) ❸
        }
        return  ❹
}
```

⓿ 可以使用命名返回值;

❶ 如果长度是零，返回 0;

❷ 否则，计算平均值;

❸ 为了使除法能正常计算，必须将值转换为 `float64`;

❹ 得到平均值，返回它

A6. (0) 整数顺序

1. 这里可以利用 Go 中的多返回值（参阅 “多值返回” 小节）:

```
func order(a, b int) (int , int) {
        if a > b {
                return b,a
        }
        return a,b
}
```

A7. (1) 作用域

1. 这个程序不能被编译，由于第 9 行的变量 `i`，未定义：`i` 仅在 `for` 循环中有效。为了修正这个，`main()` 应修改为:

```
func main() {
        var i int
        for i = 0; i < 10; i++ {
                fmt.Printf("%v\n", i)
```

```
        }
        fmt.Printf("%v\n", i)
}
```

现在 i 在 for 循环外定义，并且在其后仍然可访问。这会打印数字从 0 到 10。

A8. (1) 栈

1. 首先定义一个新的类型来表达栈；需要一个数组（来保存键）和一个指向最后一个元素的索引。这个小栈只能保存 10 个元素。

```
type stack struct {      ← 栈不应该被导出
    i    int
    data [10]int
}
```

然后需要 push 和 pop 函数来使用这个。*首先展示一下错误的解法！* 在 Go 的数据传递中，是*值传递*，意味着一个副本被创建并传递给函数。push 函数的第一个版本大约是这样：

```
func (s stack) push(k int) {      ← 工作于参数的副本
        if s.i+1 > 9 {
                return
        }
        s.data[s.i] = k
        s.i++
}
```

函数对 stack 类型的变量 s 进行处理。调用这个，只需要 s.push(50)，将整数 50 放入栈中。但是 push 函数得到的是 s 的副本，所以它不会有*真正*的结果。用这个方法，不会有内容放入栈中，例如下面的代码：

```
var s stack      ← 让 s 是一个 stack 变量
s.push(25)
fmt.Printf("stack %v\n", s);
s.push(14)
fmt.Printf("stack %v\n", s);
```

打印：

```
stack [0:0]
stack [0:0]
```

为了解决这个，需要向函数 push 提供一个指向栈的指针。这意味着需要修改 push

func (s stack)push(k int) → func (s *stack)push(k int)

应当使用 new()（参阅第 4 章 “用 new 分配内存” 小节）创建*指针*指向的 stack 的空间，因此例子中的第 1 行需要是 s := new(stack)

而两个函数变为：

```
func (s *stack) push(k int) {
        s.data[s.i] = k
        s.i++
}

func (s *stack) pop() int {
        s.i--
        return s.data[s.i]
}
```

像下面这样使用

```
func main() {
        var s stack
        s.push(25)
        s.push(14)
        fmt.Printf("stack %v\n", s)
}
```

2. 这里有一个额外的问题，对于这个练习中编写打印栈的代码的时候非常有价值。根据 Go 文档 `fmt.Printf("\%v")` 可以打印实现了 `Stringer` 接口的任何值（%v）。为了使其工作，需要为类型定义一个 `String()` 函数：

Listing 2.14. stack.String()

```
func (s stack) String() string {
        var str string
        for i := 0; i <= s.i; i++ {
                str = str + "[" +
                       strconv.Itoa(i) + ":" + strconv.Itoa
                          (s.data[i]) + "]"
        }
        return str
}
```

A9. (1) 变参

1. 需要使用 `...` 语法来实现函数接受若干个数字作为变参。

Listing 2.15. 有变参的函数

```
package main

import "fmt"

func main() {
        prtthem(1, 4, 5, 7, 4)
        prtthem(1, 2, 4)
}
```

```
func prtthem(numbers ... int) {      ← numbers 现在是整数类型的 slice
        for _, d := range numbers {
                fmt.Printf("%d\n", d)
        }
}
```

A10. (1) 斐波那契

1. 下面的程序会计算出斐波那契数列。

Listing 2.16. Go 编写的斐波那契函数

```
package main

import "fmt"

func fibonacci(value int) []int {
        x := make([]int, value) ⓿
        x[0], x[1] = 1, 1 ❶
        for n := 2; n < value; n++ {
                x[n] = x[n-1] + x[n-2] ❷
        }
        return x ❸
}

func main() {
        for _, term := range fibonacci(10) { ❹
                fmt.Printf("%v ", term)
        }
}
```

⓿ 创建一个用于保存函数执行结果的 array;

❶ 开始计算斐波那契数列;

❷ $x_n = x_{n-1} + x_{n-2}$;

❸ 返回整个 array;

❹ 使用关键字 range 可以 "遍历" 数字得到斐波那契函数返回的序列。这里有 10 个，且打印了出来。

A11. (1) map 函数

Listing 2.17. Map 函数

1.
```
func Map(f func(int) int, l []int) []int {
        j := make([]int, len(l))
        for k, v := range l {
                j[k] = f(v)
        }
```

```
        return j
}

func main() {
        m := []int{1, 3, 4}
        f := func(i int) int {
                return i * i
        }
        fmt.Printf("%v", (Map(f, m)))
}
```

2. 字符串问题的答案

A12. (0) 最小值和最大值

1. 这个函数返回 slice l 中的最大整数：

```
func max(l []int) (max int) {    ⓿
        max = l[0]
        for _, v := range l {    ❶
                if v > max {     ❷
                        max = v
                }
        }
        return  ❸
}
```

⓿ 使用了命名返回参数；

❶ 对 l 循环。元素的序号不重要；

❷ 如果找到了新的最大值，记住它；

❸ 一个 “遥远的” 返回，当前的 max 值被返回。

2. 这个函数返回 slice l 中的最小整数，这几乎与 max 完全一致。

```
func min(l []int) (min int) {
        min = l[0]
        for _, v := range l {
                if v < min {
                        min = v
                }
        }
        return
}
```

有心的读者可能已经将 max 和 min 合成一个函数，用一个选择来判断是取最小值还是最大值，或者两个值都返回。

A13. (1) 冒泡排序

1. 冒泡排序并不是最有效率的，对于 n 个元素它的算法复杂度是 $O(n^2)$。快速排序 [17] 是更好的排序算法。

   但是冒泡排序容易实现。

   Listing 2.18. 冒泡排序

   ```
   func main() {
           n := []int{5, -1, 0, 12, 3, 5}
           fmt.Printf("unsorted %v\n", n)
           bubblesort(n)
           fmt.Printf("sorted %v\n", n)
   }

   func bubblesort(n []int) {
           for i := 0; i < len(n) - 1; i++ {
                   for j := i + 1; j < len(n); j++ {
                           if n[j] < n[i] {
                                   n[i], n[j] = n[j], n[i]
                           }
                   }
           }
   }
   ```

   由于 slice 是一个引用类型，`bubblesort` 函数可以工作，并且无须返回排序后的 slice。

A14. (1) 函数返回一个函数

1. ```
   func main() {
           p2 := plusTwo()
           fmt.Printf("%v\n",p2(2))
   }

   func plusTwo() func(int) int { ⓿
           return func(x int) int { return x + 2 } ❶
   }
   ```

   ⓿ 定义新的函数返回一个函数。看看你写的跟要表达的意思是如何的；

   ❶ 函数符号，在返回语句中定义了一个 +2 的函数。

2. 这里我们使用闭包：

   ```
   func plusX(x int) func(int) int { ⓿
           return func(y int) int { return x + y } ❶
   }
   ```

   ⓿ 再次定义一个函数返回一个函数；

   ❶ 在函数符号中使用*局部*变量 `x`。
   ```

3 包

"^"

对是否有按位非的运算符的回答。
KEN THOMPSON

包是函数和数据的集合。用 package 关键字定义一个包。文件名不需要与包名一致。包名的约定是使用小写字符。Go 包可以由多个文件组成，但是使用相同的 package <name> 这一行。让我们在文件 even.go 中定义一个叫做 *even* 的包。

Listing 3.1. 一个小包

```
package even                  ← 开始自定义的包

func Even(i int) bool {       ← 可导出函数
        return i % 2 == 0
}

func odd(i int) bool {        ← 私有函数
        return i % 2 == 1
}
```

名称以大写字母起始的是*可导出*的，可以在包的外部调用（稍候会对此进行讨论）。

现在只需要构建这个包。在 $GOPATH 下建立一个目录，复制 even.go 到这个目录（参阅第 1 章的"编译和运行代码"）。

```
% mkdir $GOPATH/src/even
% cp even.go $GOPATH/src/even
% go build
% go install
```

现在就可以在程序 myeven.go 中使用这个包：

Listing 3.2. even 包的使用

```
package main

import (⓿
        "even"   ❶
        "fmt"    ❷
)

func main() {
        i := 5
        fmt.Printf("Is %d even? %v\n", i, even.Even(i)) ❸
}
```

⓿ 导入下面的包；

❶ *本地*包 *even* 在这里导入；

❷ 官方 *fmt* 包导入；

❸ 调用 *even* 包中的函数。访问一个包中的函数的语法是 `<package>.Function()`。

```
% go build myeven.go
% ./myeven
Is 5 even? false
```

在 Go 中，当函数的首字母大写的时候，函数会被从包中导出（在包外部可见，或者说公有的），因此函数名是 `Even`。如果修改 `myeven.go` 的第 10 行，使用未导出的函数 `even.odd`：

```
fmt.Printf("Is %d even? %v\n", i, even.odd(i))
```

由于使用了私有的函数，会得到一个编译错误：

```
myeven.go:10: cannot refer to unexported name even.odd
```

概括来说：

- 公有函数的名字以大写字母开头；
- 私有函数的名字以小写字母开头。

这个规则同样适用于定义在包中的其他名字（新类型、全局变量）。注意，"大写" 的含义并不仅限于 US ASCII，它被扩展到了所有大小写字母表（拉丁文、希腊文、斯拉夫文、亚美尼亚文和埃及古文）。

标识符

像在其他语言中一样，Go 的命名是很重要的。在某些情况下，它们甚至有语义上的作用：例如，在包外是否可见决定于首字母是不是大写。因此有必要花点时间讨论一下 Go 程序的命名规则。

使用的规则是让众所周知的缩写保持原样，而不是去尝试到底哪里应该大写。`Atoi`，`Getwd`，`Chmod`。

驼峰式对那些有完整单词的会很好：`ReadFile`，`NewWriter`，`MakeSlice`。

包名

当包导入（通过 `import`）时，包名成为了内容的入口。在

```
import "bytes"
```

之后，导入包的可以调用函数 `bytes.Buffer`。任何使用这个包的人，可以使用同样的名字访问到它的内容，因此这样的包名是好的：短的、简洁的、好记的。根据规则，包名是小写的一个单词；不应当有下划线或混合大小写。保持简洁（由于每个人都可能需要录入这个名字），不要过早考虑命名冲突。

包名是导入的默认名称。可以通过在导入语句指定其他名称来覆盖默认名称：

```
import bar "bytes"
```

函数 `Buffer` 现在可以通过 `bar.Buffer` 来访问。这意味着，包名无需全局唯一；在少有的冲突中，可以给导入的包选择另一个名字在局部使用。在任何时候，冲突都是很少见的，因为导入的文件名会用来做判断，到底是哪个包使用了。

另一个规则是包名就是代码的根目录名；在 *src/pkg/compress/gzip* 的包，作为 `compress/gzip` 导入，但名字是 *gzip*，不是 *compress_gzip* 也不是 *compressGzip*。

导入包将使用其名字引用到内容上，所以导入的包可以利用这个避免罗嗦。例如，缓冲类型 *bufio* 包的读取方法，叫做 `Reader`，而不是 `BufReader`，因为用户看到的是 `bufio.Reader` 这个清晰、简洁的名字。更进一步说，由于导入的实例总是它们包名指向的地址，`bufio.Reader` 不会与 `io.Reader` 冲突。类似的，`ring.Ring`（包 *container/ring*）创建新实例的函数——在 Go 中定义的构造函数——通常叫做 `NewRing`，但是由于 `Ring` 是这个包唯一的一个导出的类型，同时，这个包也叫做 *ring*，所以它可以只称作 `New`。包的客户看到的是 `ring.New`。用包的结构帮助你选择更好的名字。

另外一个简短的例子是 `once.Do`（参看 *sync*）；`once.Do(setup)` 读起来很不错，并且命名为 `once.DoOrWaitUntilDone(setup)` 不会有任何帮助。长的名字不会让其变得容易阅读。如果名字表达了一些复杂并且微妙的内容，更好的办法是编写一些有帮助的注释，而不是将所有信息都放入名字里。

最后，在 Go 中使用混合大小写 MixedCaps 或者 mixedCaps，而不是下划线区分含有多个单词的名字。

包的文档

这段复制于 [8]。

每个包都应该有*包注释*，在 `package` 前的一个注释块。对于多文件包，包注释只需要出现在一个文件前，任意一个文件都可以。包注释应当对包进行介绍，并提供相关于包的整体信息。这会出现在 `go doc` 生成的关于包的页面上，并且相关的细节会一并显示。来自官方 *regexp* 包的例子：

```
/*
    The regexp package implements a simple library for
    regular expressions.

    The syntax of the regular expressions accepted is:

    regexp:
        concatenation  '|' concatenation
*/
package regexp
```

每个定义（并且导出）的函数应当有一小段文字描述该函数的行为。来自于 *fmt* 包的例子：

```
// Printf formats according to a format specifier and writes to standard
// output. It returns the number of bytes written and any write error
// encountered.
func Printf(format string, a ...interface) (n int, err error)
```

测试包

在 Go 中为包编写单元测试应当是一种习惯。编写测试需要包含 *testing* 包和程序 `go test`。两者都有良好的文档。

`go test` 程序调用了所有的测试函数。*even* 包没有定义任何测试函数，执行 `go test`，这样：

```
% go test
?       even    [no test files]
```

在测试文件中定义一个测试来修复这个。测试文件也在包目录中，被命名为 `*_test.go`。这些测试文件同 Go 程序中的其他文件一样，但是 `go test` 只会执行测试函数。每个测试函数都有相同的标识，它的名字以 `Test` 开头：

```
func TestXxx(t *testing.T)
```

编写测试时，需要告诉 `go test` 测试是失败还是成功。测试成功则直接返回。当测试失败可以用下面的函数标记 [11]。这是非常重要的（参阅 `go doc testing` 或 `go help testfunc` 了解更多）：

```
func (t *T) Fail()
```

`Fail` 标记测试函数失败，但仍然继续执行。

```
func (t *T) FailNow()
```

`FailNow` 标记测试函数失败，并且中断其执行。当前文件中的其余的测试将被跳过，然后执行下一个文件中的测试。

```
func (t *T) Log(args ...interface{})
```

`Log` 用默认格式对其参数进行格式化，与 `Print()` 类似，并且记录文本到错误日志。

```
func (t *T) Fatal(args ...interface{})
```

`Fatal` 等价于 `Log()` 后跟随 `FailNow()`。

将这些凑到一起，就可以编写测试了。首先，选择名字 `even_test.go`。然后添加下面的内容：

Listing 3.3. even 包的测试

```
package even

import "testing"

func TestEven(t *testing.T) {
        if ! Even(2) {
                t.Log("2 should be even!")
                t.Fail()
        }
}
```

注意在第一行使用了 package even，测试使用与被测试的包使用相同的名字空间。这不仅仅是为了方便，也允许了测试未导出的函数和结构。然后导入 *testing* 包，并且在第 5 行定义了这个文件中唯一的测试函数。展示的 Go 代码应当没有任何惊异的地方：检查了 Even 函数是否工作正常。现在等待了好久的时刻到了，执行测试：

```
% go test
ok      even    0.001s
```

测试执行并且报告 ok。成功了！

如果重新定义测试函数，就可以看到一个失败的测试：

```
// Entering the twilight zone
func TestEven(t *testing.T) {
        if Even(2) {
                t.Log("2 should be odd!")
                t.Fail()
        }
}
```

然后得到：

```
FAIL    even    0.004s
--- FAIL: TestEven (0.00 seconds)
       2 should be odd!
FAIL
```

然后你可以以此行事（修复测试的实例）

> 在编写包的时候应当一边写代码，一边写（一些）文档和测试函数。这可以让你的程序更好，并且它展示了你的努力。

The Go test suite also allows you to incorperate example functions which serve as documentation *and* as tests. These functions need to start with Example.

```
func ExampleEven() {
    if Even(2) {
        fmt.Printf("Is even\n")
    }
    // Output:
    // Is even
}
```

Those last two comments lines are part of the example, go test uses those to check the *generated* output with the text in the comments. If there is a mismatch the test fails.

常用的包

标准的 Go 代码库中包含了大量的包，并且在安装 Go 的时候多数会伴随一起安装。浏览 $GOROOT/src/pkg 目录并且查看那些包会非常有启发。无法对每个包就加以解说，不过下面的这些值得讨论：[a]

[a]描述来自包的 go doc。额外的解释用斜体。

fmt

包 *fmt* 实现了格式化的 I/O 函数，这与 C 的 `printf` 和 `scanf` 类似。格式化短语派生于 C 。一些短语（%-序列）这样使用：

%v

默认格式的值。当打印结构时，加号（%+v）会增加字段名；

%#v

Go 样式的值表达；

%T

带有类型的 Go 样式的值表达；

io

这个包提供了原始的 I/O 操作界面。它主要的任务是对 os 包这样的原始的 I/O 进行封装，增加一些其他相关，使其具有抽象功能用在公共的接口上。

bufio

这个包实现了缓冲的 I/O。它封装于 `io.Reader` 和 `io.Writer` 对象，创建了另一个对象（Reader 和 Writer）在提供缓冲的同时实现了一些文本 I/O 的功能。

sort

sort 包提供了对数组和用户定义集合的原始的排序功能。

strconv

strconv 包提供了将字符串转换成基本数据类型，或者从基本数据类型转换为字符串的功能。

os

os 包提供了与平台无关的操作系统功能接口。其设计是 Unix 形式的。

sync

sync 包提供了基本的同步原语，例如互斥锁。

flag

flag 包实现了命令行解析。*参阅 "命令行参数" 在第 92 页。*

encoding/json

encoding/json 包实现了编码与解码 RFC 4627 [2] 定义的 JSON 对象。

html/template

数据驱动的模板，用于生成文本输出，例如 HTML。

将模板关联到某个数据结构上进行解析。模板内容指向数据结构的元素（通常结构的字段或者 map 的键）控制解析并且决定某个值会被显示。模板扫描结构以便解析，而 "游标" @ 决定了当前位置在结构中的值。

net/http

net/http 实现了 HTTP 请求、响应和 URL 的解析，并且提供了可扩展的 HTTP 服务和基本的 HTTP 客户端。

unsafe

unsafe 包包含了 Go 程序中数据类型上所有不安全的操作。*通常无须使用这个。*

reflect

reflect 包实现了运行时反射，允许程序通过抽象类型操作对象。通常用于处理静态类型 interface{} 的值，并且通过 `Typeof` 解析出其动态类型信息，通常会返回一个有接口类型 `Type` 的对象。

参阅 5，第 "自省和反射" 节。

os/exec

os/exec 包执行外部命令。

练习

Q15. (0) stack 包

1. 参考 Q8 练习。在这个练习中将从那个代码中建立一个独立的包。为 stack 的实现创建一个合适的包，`Push`、`Pop` 和 `Stack` 类型需要被导出。
2. 为这个包编写一个单元测试，至少测试 `Push` 后 `Pop` 的工作情况。

Q16. (2) 计算器

1. 使用 stack 包创建逆波兰计算器。

答案

A15. (0) stack 包

1. 在创建 stack 包时，仅有一些小细节需要修改。首先，导出的函数应当大写首字母，因此应该是 `Stack`。包所在的文件被命名为 `stack-as-package.go`，内容是：

   Listing 3.4. 包里的 Stack

```go
package stack

// 保存元素的 Stack
type Stack struct {
	i    int
	data [10]int
}

// Push 将元素压入栈中
func (s *Stack) Push(k int) {
	s.data[s.i] = k
	s.i++
}

// Pop 从栈中弹出一个元素
func (s *Stack) Pop() (ret int) {
	s.i--
	ret = s.data[s.i]
	return
}
```

2. 为了让单元测试正常工作，需要做一些准备。下面用一分钟的时间来做这些。首先是单元测试本身。创建文件 `pushpop_test.go`，有如下内容：

   Listing 3.5. Push/Pop 测试

```go
package stack

import "testing"

func TestPushPop(t *testing.T) {
	c := new(Stack)
	c.Push(5)
	if c.Pop() != 5 {
		t.Log("Pop doesn't give 5")
		t.Fail()
	}
}
```

为了让 `go test` 能够工作，需要将包所在文件放到 `$GOPATH/src`：

```
% mkdir $GOPATH/src/stack
% cp pushpop_test.go $GOPATH/src/stack
% cp stack-as-package.go $GOPATH/src/stack
```

输出：

```
% go test stack
ok      stack   0.001s
```

A16. (2) 计算器

1. 这是第一个答案：

Listing 3.6. 逆波兰计算器

```
package main

import (
        "bufio"
        "fmt"
        "os"
        "strconv"
)

var reader *bufio.Reader = bufio.NewReader(os.Stdin)
var st = new(Stack)

type Stack struct {
        i    int
        data [10]int
}

func (s *Stack) push(k int) {
        if s.i+1 > 9 {
                return
        }
        s.data[s.i] = k
        s.i++
}

func (s *Stack) pop() (ret int) {
        s.i--
        if s.i < 0 {
                s.i = 0
                return
        }
        ret = s.data[s.i]
        return
```

```
}

func main() {
        for {
                s, err := reader.ReadString('\n')
                var token string
                if err != nil {
                        return
                }
                for _, c := range s {
                        switch {
                        case c >= '0' && c <= '9':
                                token = token + string(c)
                        case c == ' ':
                                r, _ := strconv.Atoi(token)
                                st.push(r)
                                token = ""
                        case c == '+':
                                fmt.Printf("%d\n", st.pop()+
                                    st.pop())
                        case c == '*':
                                fmt.Printf("%d\n", st.pop()*
                                    st.pop())
                        case c == '-':
                                p := st.pop()
                                q := st.pop()
                                fmt.Printf("%d\n", q-p)
                        case c == 'q':
                                return
                        default:
                                //error
                        }
                }
        }
}
```

4 进阶

> “Go 有指针，但是没有指针运算。你不能用指针变量遍历字符串的各个字节。”
>
> *Go For C++ Programmers*
> GO AUTHORS

Go 有指针。然而却没有指针运算，因此它们更象是引用而不是你所知道的来自于 C 的指针。指针非常有用。在 Go 中调用函数的时候，得记得变量是*值传递*的。因此，为了修改一个传递入函数的值的效率和可能性，有了指针。

通过类型作为前缀来定义一个指针’*’：var p *int。现在 p 是一个指向整数值的指针。所有新定义的变量都被赋值为其类型的零值，而指针也一样。一个新定义的或者没有任何指向的指针，有值 nil。在其他语言中，这经常被叫做空（NULL）指针，在 Go 中就是 nil。让指针指向某些内容，可以使用取址操作符 （&），像这样：

Listing 4.1. 指针的使用

```
var p *int
fmt.Printf("%v", p)        ← 打印 nil

var i int                  ← 定义一个整形变量 i
p = &i                     ← 使得 p 指向 i

fmt.Printf("%v", p)        ← 打印出来的内容类似 0x7ff96b81c000a
```

从指针获取值是通过在指针变量前置’*’ 实现的：

Listing 4.2. 获取指针指向的值

```
p = &i                     ← 获取 i 的地址
*p = 8                     ← 修改 i 的值
fmt.Printf("%v\n", *p)     ← 打印 8
fmt.Printf("%v\n", i)      ← 同上
```

前面已经说了，没有指针运算，所以如果这样写：*p++，它表示 (*p)++：首先获取指针指向的值，然后对这个值加一。[a]

内存分配

Go 同样也垃圾收集，也就是说无须担心内存分配和回收。

Go 有两个内存分配原语，new 和 make。它们应用于不同的类型，做不同的工作，可能有些迷惑人，但是规则很简单。下面的章节展示了在 Go 中如何处理内存分配，并且希望能够让 new 和 make 之间的区别更加清晰。

---

[a] 参看练习 17。

用 new 分配内存

内建函数 new 本质上说跟其他语言中的同名函数功能一样：new(T) 分配了零值填充的 T 类型的内存空间，并且返回其地址，一个 *T 类型的值。用 Go 的术语说，它返回了一个指针，指向新分配的类型 T 的零值。记住这点非常重要。

这意味着使用者可以用 new 创建一个数据结构的实例并且可以直接工作。如 bytes.Buffer 的文档所述 "Buffer 的零值是一个准备好了的空缓冲。" 类似的，sync.Mutex 也没有明确的构造函数或 Init 方法。取而代之，sync.Mutex 的零值被定义为非锁定的互斥量。

零值是非常有用的。例如这样的类型定义，57 页的"定义自己的类型" 内容。

```
type SyncedBuffer struct {
    lock    sync.Mutex
    buffer  bytes.Buffer
}
```

SyncedBuffer 的值在分配内存或定义之后立刻就可以使用。在这个片段中，p 和 v 都可以在没有任何更进一步处理的情况下工作。

```
p := new(SyncedBuffer)        ← Type *SyncedBuffer，已经可以使用
var v SyncedBuffer            ← Type SyncedBuffer，同上
```

用 make 分配内存

回到内存分配。内建函数 make(T, args) 与 new(T) 有着不同的功能。它*只能*创建 slice，map 和 channel，并且返回一个有初始值（非零）的 T 类型，而不是 *T。本质来讲，导致这三个类型有所不同的原因是指向数据结构的引用在使用前必须被初始化。例如，一个 slice，是一个包含指向数据（内部 array）的指针，长度和容量的三项描述符；在这些项目被初始化之前，slice 为 nil。对于 slice，map 和 channel，make 初始化了内部的数据结构，填充适当的值。

例如，make([]int, 10, 100) 分配了 100 个整数的数组，然后用长度 10 和容量 100 创建了 slice 结构指向数组的前 10 个元素。区别是，new([]int) 返回指向新分配的内存的指针，而零值填充的 slice 结构是指向 nil 的 slice 值。

这个例子展示了 new 和 make 的不同。

```
var p *[]int = new([]int)       ← 分配 slice 结构内存；很少使用
var v  []int = make([]int, 100) ← v 指向一个新分配的有 100 个整数的数组

    var p *[]int = new([]int)       ← 不必要的复杂例子
*p = make([]int, 100, 100)

    v := make([]int, 100)           ← 更常见
```

务必记得 make 仅适用于 map，slice 和 channel，并且返回的不是指针。应当用 new 获得特定的指针。

> new 分配；make 初始化
>
> 上面的两段可以简单总结为：
>
> - new(T) 返回 *T 指向一个零值 T
> - make(T) 返回初始化后的 T
>
> 当然 make 仅适用于 slice，map 和 channel。

构造函数与复合声明

有时零值不能满足需求，必须要有一个用于初始化的构造函数，例如这个来自 *os* 包的例子。

```
func NewFile(fd int, name string) *File {
    if fd < 0 {
        return nil
    }
    f := new(File)
    f.fd = fd
    f.name = name
    f.dirinfo = nil
    f.nepipe = 0
    return f
}
```

有许多冗长的内容。可以使用复合声明使其更加简洁，每次只用一个表达式创建一个新的实例。

```
func NewFile(fd int, name string) *File {
    if fd < 0 {
        return nil
    }
    f := File{fd, name, nil, 0}     ← Create a new File
    return &f     ← 返回 f 的地址
}
```

返回本地变量的地址没有问题；在函数返回后，相关的存储区域仍然存在。

事实上，从复合声明获取分配的实例的地址更好，因此可以最终将两行缩短到一行。[b]

```
return &File{fd, name, nil, 0}
```

The items (called of a composite +literal are laid out in order and must all be 所有的项目（称作 字段）都必须按顺序全部写上。然而，通过对元素用字段: 值成对的标识，初始化内容可以按任意顺序出现，并且可以省略初始化为零值的字段。因此可以这样

```
return &File{fd: fd, name: name}
```

[b]从复合声明中获取地址，意味着告诉编译器在堆中分配空间，而不是栈中。

在特定的情况下，如果复合声明不包含任何字段，它创建特定类型的零值。表达式 new(File) 和 &File{} 是等价的。

复合声明同样可以用于创建 array，slice 和 map，通过指定适当的索引和 map 键来标识字段。在这个例子中，无论是 Enone，Eio 还是 Einval 初始化都能很好的工作，只要确保它们不同就好了。

```
ar := [...]string{Enone: "no error", Einval: "invalid argument"}
sl := []string{Enone: "no error", Einval: "invalid argument"}
ma := map[int]string {Enone: "no error", Einval: "invalid argument
    "}
```

定义自己的类型

自然，Go 允许定义新的类型，通过关键字 type 实现：

```
type foo int
```

创建了一个新的类型 foo 作用跟 int 一样。创建更加复杂的类型需要用到 struct 关键字。这有个在一个数据结构中记录某人的姓名（string）和年龄（int），并且使其成为一个新的类型的例子：

Listing 4.3. 结构体

```
package main

import "fmt"

type NameAge struct {
        name string        ← 不导出
        age  int           ← 不导出
}

func main() {
        a := new(NameAge)
        a.name = "Pete"; a.age = 42

        fmt.Printf("%v\n", a)
}
```

通常，fmt.Printf("%v\n", a) 的输出是

```
&{Pete 42}
```

这很棒！Go 知道如何打印结构。如果仅想打印某一个，或者某几个结构中的字段，需要使用 .<field name>。例如，仅仅打印名字：

```
fmt.Printf("%s", a.name)     ← %s 格式化字符串
```

结构字段

之前已经提到结构中的项目被称为field。没有字段的结构：struct {}
或者有四个[c] 字段的：

```
struct {
        x, y int
        A *[]int
        F func()
}
```

如果省略字段的名字，可以创建匿名字段，例如：

```
struct {
        T1            ← 字段名字是 T1
        *T2           ← 字段名字是 T2
        P.T3          ← 字段名字是 T3
        x, y int      ← 字段名字是 x 和 y
}
```

注意首字母大写的字段可以被导出，也就是说，在其他包中可以进行读写。字段名以小写字母开头是当前包的私有的。包的函数定义是类似的，参阅第 3 章了解更多细节。

方法

可以对新定义的类型创建函数以便操作，可以通过两种途径：

1. 创建一个函数接受这个类型的参数。

   ```
   func doSomething(n1 *NameAge, n2 int) { /* */ }
   ```

   （你可能已经猜到了）这是 *函数调用*。

2. 创建一个工作在这个类型上的函数（参阅在 2.1 中定义的*接收方*）：

   ```
   func (n1 *NameAge) doSomething(n2 int) { /* */ }
   ```

   这是*方法调用*，可以类似这样使用：

   ```
   var n *NameAge
   n.doSomething(2)
   ```

使用函数还是方法是由程序员决定的，但是如果想要满足接口（参阅下一章）就只能使用方法。如果没有这方面的需求，那就由个人品味决定了。

使用函数还是方法完全是由程序员说了算，但是若需要满足接口（参看下一章）就必须使用方法。如果没有这样的需求，那就完全由习惯来决定是使用函数还是方法了。

但是下面的内容一定要留意，引用自 [10]：

> *如果 x 可获取地址，并且 &x 的方法中包含了 m，x.m() 是 (&x).m() 更短的写法。*

[c]是的，四（4）个。

根据上面所述，这意味着下面的情况不是错误：

```
var n NameAge          ← 不是指针
n.doSomething(2)
```

这里 Go 会查找 `NameAge` 类型的变量 `n` 的方法列表，没有找到就会*再*查找 `*NameAge` 类型的方法列表，并且将其转化为 `(&n).doSomething(2)`。

下面的类型定义中有一些微小但是很重要的不同之处。同时可以参阅 [10, section "Type Declarations"]。假设有：

```
// Mutex 数据类型有两个方法，Lock 和 Unlock。
type Mutex struct          { /* Mutex 字段 */ }
func (m *Mutex) Lock()     { /* Lock 实现 */ }
func (m *Mutex) Unlock()   { /* Unlock 实现 */ }
```

现在用两种不同的风格创建了两个数据类型。

- type `NewMutex Mutex`;
- type `PrintableMutex` struct {`Mutex` }.

现在 `NewMutux` 等同于 `Mutex`，但是它没有任何 `Mutex` 的方法。换句话说，它的方法是空的。

但是 `PrintableMutex` *已经从* `Mutex` *继承*了方法集合。如同 [10] 所说：

> `*PrintableMutex` *的方法集合包含了* `Lock` *和* `Unlock` *方法，被绑定到其匿名字段* `Mutex`。

转换

有时需要将一个类型转换为另一个类型。在 Go 中可以做到，不过有一些规则。首先，将一个值转换为另一个是由操作符（看起来像函数：`byte()`）完成的，并且不是所有的转换都是允许的。

*Table 4.1. 合法的转换，*float64 *同* float32 *类似。注意，为了适配表格的显示，float32 被简写为 flt32。*

From / To	b []byte	i []int	r []rune	s string	f flt32	i int
[]byte	×			[]byte(s)		
[]int		×		[]int(s)		
[]rune			×	[]rune(s)		
string	string(b)	string(i)	string(r)	×		
ftl32					×	flt32(i)
int					int(f)	×

- 从 string 到字节或者 ruin 的 slice。

```
mystring := "hello this is string"
```

```
byteslice := []byte(mystring)
```

转换到 byte slice，每个 byte 保存字符串对应字节的整数值。注意 Go 的字符串是 UTF-8 编码的，一些字符可能是 1、2、3 或者 4 个字节结尾。

```
runeslice  := []rune(mystring)
```

转换到 rune slice，每个 rune 保存 Unicode 编码的指针。字符串中的每个字符对应一个整数。

- 从字节或者整形的 slice 到 string。

```
b := []byte{'h','e','l','l','o'} // 复合声明
s := string(b)
i := []rune{257,1024,65}
r := string(i)
```

对于数值，定义了下面的转换：

- 将整数转换到指定的（bit）长度：uint8(int)；
- 从浮点数到整数：int(float32)。这会截断浮点数的小数部分；
- 其他的类似：float32(int)。

用户定义类型的转换

如何在自定义类型之间进行转换？这里创建了两个类型 Foo 和 Bar，而 Bar 是 Foo 的一个别名：

```
type foo struct { int }      ← 匿名字段
type bar foo                 ← bar 是 foo 的别名
```

然后：

```
var b bar = bar{1}      ← 声明 b 为 bar 类型
var f foo = b           ← 赋值 b 到 f
```

最后一行会引起错误：

cannot use b (type bar) as type foo in assignment(不能使用 b（类型 bar）作为类型 foo 赋值)

这可以通过转换来修复：

```
var f foo = foo(b)
```

注意转换那些字段不一致的结构是相当困难的。同时注意，转换 b 到 int 同样会出错；整数与有整数字段的结构并不一样。

组合

TODO(miek):work in progress Go 不是面向对象语言，因此并没有继承。但是有时又会需要从已经实现的类型中“继承”并修改一些方法。在 Go 中可以用嵌入一个类型的方式来实现。

练习

Q17. (1) 指针运算

1. 在正文的第 54 页有这样的文字：

   > *…这里没有指针运算，因此如果这样写：*`*p++`*，它被解释为* `(*p)++`*：首先解析引用然后增加值。*

   当像这样增加一个值的时候，什么类型可以工作？

2. 为什么它不能工作在所有类型上？

Q18. (2) 使用 interface 的 map 函数

1. 使用练习 Q11 的答案，利用 interface 使其更加通用。让它至少能同时工作于 int 和 string。

Q19. (1) 指针

1. 假设定义了下面的结构：

   ```
   type Person struct {
       name string
       age  int
   }
   ```

   下面两行之间的区别是什么？

   ```
   var p1 Person
   p2 := new(Person)
   ```

2. 下面两个内存分配的区别是什么？

   ```
   func Set(t *T) {
       x = t
   }
   ```

   和

   ```
   func Set(t T) {
       x= &t
   }
   ```

Q20. (1) Linked List

1. Make use of the package *container/list* to create a (doubly) linked list. Push the values 1, 2 and 4 to the list and then print it.

2. Create your own linked list implementation. And perform the same actions as in question 1

Q21. (1) Cat

1. 编写一个程序，模仿 Unix 的 `cat` 程序。对于不知道这个程序的人来说，下面的调用显示了文件 `blah` 的内容：

   ```
   % cat blah
   ```

2. 使其支持 n 开关，用于输出每行的行号。
3. 上面问题中，1 提供的解决方案存在一个 Bug。你能定位并修复它吗？

Q22. (2) 方法调用

1. 假设有下面的程序。要注意的是包 *container/vector* 曾经是 Go 的一部分，但是当内建的 append 出现后，就被移除了。然而，对于当前的问题这不重要。这个包实现了有 push 和 pop 方法的栈结构。

   ```
   package main

   import "container/vector"

   func main() {
           k1 := vector.IntVector{}
           k2 := &vector.IntVector{}
           k3 := new(vector.IntVector)
           k1.Push(2)
           k2.Push(3)
           k3.Push(4)
   }
   ```

   `k1`，`k2` 和 `k3` 的类型是什么？

2. 当前，这个程序可以编译并且运行良好。在不同类型的变量上 `Push` 都可以工作。`Push` 的文档这样描述：

   > *func (p *IntVector) Push(x int) Push 增加 x 到向量的末尾。*

   那么接受者应当是 `*IntVector` 类型，为什么上面的代码（Push 语句）可以正确工作？above (the Push statements) work correct then?

答案

A17. (1) 指针运算

1. 这仅能工作于指向数字（`int`，`uint` 等等）的指针值。
2. `++` 仅仅定义在数字类型上，同时由于在 Go 中没有运算符重载，所以会在其他类型上失败（编译错误）。

A18. (2) 使用 interface 的 map 函数

Listing 4.4. Go 中更加通用的 map 函数

```
package main

import "fmt"

//* define the empty interface as a type
type e interface{}

func mult2(f e) e {
        switch f.(type) {
        case int:
                return f.(int) * 2
        case string:
                return f.(string) + f.(string) + f.(string)
                   + f.(string)
        }
        return f
}

func Map(n []e, f func(e) e) []e {
        m := make([]e, len(n))
        for k, v := range n {
                m[k] = f(v)
        }
        return m
}

func main() {
        m := []e{1, 2, 3, 4}
        s := []e{"a", "b", "c", "d"}
        mf := Map(m, mult2)
        sf := Map(s, mult2)
        fmt.Printf("%v\n", mf)
        fmt.Printf("%v\n", sf)
}
```

A19. (1) 指针

1. 第一行：`var p1 Person` 分配了 `Person`-*值* 给 `p1`。`p1` 的类型是 `Person`。

   第二行：`p2 := new(Person)` 分配了内存并且将*指针*赋值给 `p2`。`p2` 的类型是 `*Person`。

2. 在第二个函数中，`x` 指向一个新的（堆上分配的）变量 `t`，其包含了实际参数值的副本。

   在第一个函数中，`x` 指向了 `t` 指向的内容，也就是实际上的参数指向的内容。

   因此在第二个函数，我们有了 "额外" 的变量存储了相关值的副本。

A20. (1) Linked List

1. The following is the implementation of a program using doubly linked lists from *container/list*.

   Listing 4.5. Doubly linked list using container/list

```
package main

import (
        "fmt"
        "container/list"
)

func main() {
        l := list.New()
        l.PushBack(1)
        l.PushBack(2)
        l.PushBack(4)

        for e := l.Front(); e != nil; e = e.Next() {
                fmt.Printf("%v\n", e.Value)
        }
}
```

2. The following is a program implementing a simple doubly linked list supporting `int` values.

   Listing 4.6. Doubly linked list

```
package main

⓿
import (
        "errors"
        "fmt"
)

type Value int ❶
```

```
type Node struct { ❷
        Value
        prev, next *Node
}

type List struct {
        head, tail *Node
}

❸
func (l *List) Front() *Node {
        return l.head
}

func (n *Node) Next() *Node {
        return n.next
}

func (l *List) Push(v Value) *List {
        n := &Node{Value: v} ❹

        if l.head == nil { ❺
                l.head = n
        } else {
                l.tail.next = n ❻
                n.prev = l.tail ❼
        }
        l.tail = n ❽

        return l
}

var errEmpty = errors.New("List is empty")

func (l *List) Pop() (v Value, err error) {
        if l.tail == nil { ❾
                err = errEmpty
        } else {
                v = l.tail.Value ❿
                l.tail = l.tail.prev ⓫
                if l.tail == nil {
                        l.head = nil ⓬
                }
        }
```

```
        return v, err
}

func main() {
        l := new(List)

        l.Push(1)
        l.Push(2)
        l.Push(4)

        for n := l.Front(); n != nil; n = n.Next() {
                fmt.Printf("%v\n", n.Value)
        }

        fmt.Println()

        for v, err := l.Pop(); err == nil; v, err = l.Pop()
            {
                fmt.Printf("%v\n", v)
        }
}
```

⓿ Include all the packages we need.

❶ Declare a type for the value our list will contain;

❷ declare a type for the each node in our list;

❸ Mimic the interface of container/list.

❹ When pushing, create a new Node with the provided value;

❺ if the list is empty, put the new node at the head;

❻ otherwise put it at the tail;

❼ make sure the new node points back to the previously existing one;

❽ point tail to the newly inserted node.

❾ When popping, return an error if the list is empty;

❿ otherwise save the last value;

⓫ discard the last node from the list;

⓬ and make sure the list is consistent if it becomes empty;

A21. (1) Cat

1. 下面是 `cat` 的实现，同样支持 n 输出每行的行号。

Listing 4.7. cat 程序

```
package main
```

```
⓿
import (
        "io"
        "os"
        "fmt"
        "bufio"
        "flag"
)

var numberFlag = flag.Bool("n", false, "number each line") ❶

❷
func cat(r *bufio.Reader) {
        i := 1
        for {
                buf, e := r.ReadBytes('\n')     ❸
                if e == io.EOF {                ❹
                        break
                }
                if *numberFlag {                ❺
                        fmt.Fprintf(os.Stdout, "%5d  %s", i,
                             buf)
                        i++
                } else {                        ❻
                        fmt.Fprintf(os.Stdout, "%s", buf)
                }
        }
        return
}

func main() {
        flag.Parse()
        if flag.NArg() == 0 {
                cat(bufio.NewReader(os.Stdin))
        }
        for i := 0; i < flag.NArg(); i++ {
                f, e := os.Open(flag.Arg(i))
                if e != nil {
                        fmt.Fprintf(os.Stderr, "%s: error
                            reading from %s: %s\n",
                                os.Args[0], flag.Arg(i), e.
                                    Error())
                        continue
                }
                cat(bufio.NewReader(f))
```

```
        }
}
```

❿ 包含所有需要用到的包；

❶ 定义新的开关"n"，默认是关闭的。注意很容易写的帮助文本；

❷ 实际上读取并且显示文件内容的函数；

❸ 每次读一行；

❹ 如果到达文件结尾；

❺ 如果设定了行号，打印行号然后是内容本身；

❻ 否则，仅仅打印该行内容。

2. 当最后一行不包括换行符时，这个 Bug 就会出现。更糟糕的情况是，当输入只有一行且没有换行符的时候，什么也不显示。下面的程序是一个更好的解决方案。

Listing 4.8. 一个更好的 cat 程序

```
package main

import (
        "bufio"
        "flag"
        "fmt"
        "io"
        "os"
)

var numberFlag = flag.Bool("n", false, "number each line")

func cat(r *bufio.Reader) {
        i := 1
        for {
                buf, e := r.ReadBytes('\n')
                if e == io.EOF {
                        break
                }
                if *numberFlag {
                        fmt.Fprintf(os.Stdout, "%5d  %s", i,
                             buf)
                        i++
                } else {
                        fmt.Fprintf(os.Stdout, "%s", buf)
                }
        }
        return
}
```

```
func main() {
	flag.Parse()
	if flag.NArg() == 0 {
		cat(bufio.NewReader(os.Stdin))
	}
	for i := 0; i < flag.NArg(); i++ {
		f, e := os.Open(flag.Arg(i))
		if e != nil {
			fmt.Fprintf(os.Stderr, "%s: error
			    reading from %s: %s\n",
			        os.Args[0], flag.Arg(i), e.
			            Error())
			continue
		}
		cat(bufio.NewReader(f))
	}
}
```

A22. (2) 方法调用

1. k1 的类型是 vector.IntVector。为什么？这里使用了符号 {}，因此获得了类型的值。变量 k2 是 *vector.IntVector，因为获得了复合语句的地址（&）。而最后的 k3 同样是 *vector.IntVector 类型，因为 new 返回该类型的指针。

2. 在 [10] 的 "调用" 章节，有这样的描述：

   > *当 x 的方法集合包含 m，并且参数列表可以赋值给 m 的参数，方法调用 x.m() 是合法的。如果 x 可以被地址化，而 &x 的方法集合包含 m，x.m() 可以作为 (&x).m() 的省略写法。*

   换句话说，由于 k1 可以被地址化，而 *vector.IntVector *具有* Push 方法，调用 k1.Push(2) 被 Go 转换为 (&k1).Push(2) 来使型系统愉悦（也使你愉悦——现在你已经了解到这一点）。[d]

[d]参阅本章的第 "方法" 节。

5 接口

我对外科手术般进入我的身体总是有恐惧。你知道我说的是什么。

eXistenZ
TED PIKUL

下面的内容来自[22]。是 Ian Lance Taylor 编写的，他是 Go 的作者之一。

在 Go 中，关键字 *interface* 被赋予了多种不同的含义。每个类型都有接口，意味着对那个类型定义了方法集合。这段代码定义了具有一个字段和两个方法的结构类型 S。

Listing 5.1. 定义结构和结构的方法

```
type S struct { i int }
func (p *S) Get() int { return p.i }
func (p *S) Put(v int) { p.i = v }
```

也可以定义接口类型，仅仅是方法的集合。这里定义了一个有两个方法的接口 `I`：

```
type I interface {
  Get() int
  Put(int)
}
```

对于接口 `I`，`S` 是合法的*实现*，因为它定义了 `I` 所需的两个方法。注意，即便是没有明确定义 `S` 实现了 `I`，这也是正确的。

Go 程序可以利用这个特点来实现接口的另一个含义，就是 接口值：

```
func f(p I) {  ⓿
    fmt.Println(p.Get()) ❶
    p.Put(1) ❷
}
```

⓿ 定义一个函数接受一个接口类型作为参数；

❶ p 实现了接口 `I`，*必须*有 `Get()` 方法；

❷ `Put()` 方法是类似的。

这里的变量 p 保存了接口类型的值。因为 `S` 实现了 `I`，可以调用 `f` 向其传递 `S` 类型的值的指针：

```
var s S; f(&s)
```

获取 s 的地址，而不是 `S` 的值的原因，是因为在 s 的指针上定义了方法，参阅上面的代码 5.1。这并不是必须的——可以定义让方法接受值——但是这样的话 `Put` 方法就不会像期望的那样工作了。

实际上，无须明确一个类型是否实现了一个接口意味着 Go 实现了叫做 duck typing[26] 的模式。这不是纯粹的 duck typing，因为如果可能的话 Go 编译器将对类型是否实现

了接口进行实现静态检查。然而，Go 确实有纯粹动态的方面，如可将一个接口类型转换到另一个。通常情况下，转换的检查是在运行时进行的。如果是非法转换——当在已有接口值中存储的类型值不匹配将要转换到的接口——程序会抛出运行时错误。

在 Go 中的接口有着与许多其他编程语言类似的思路：C++ 中的纯抽象虚基类，Haskell 中的 typeclasses 或者 Python 中的 duck typing。然而没有其他任何一个语言联合了接口值、静态类型检查、运行时动态转换，以及无须明确定义类型适配一个接口。这些给 Go 带来的结果是，强大、灵活、高效和容易编写的。

到底是什么？

来定义另外一个类型同样实现了接口 `I`：

```
type R struct { i int }
func (p *R) Get() int { return p.i }
func (p *R) Put(v int) { p.i = v }
```

函数 `f` 现在可以接受类型为 `R` 或 `S` 的变量。假设需要在函数 `f` 中知道实际的类型。在 Go 中可以使用 type switch 得到。

```
func f(p I) {
    switch t := p.(type) { ⓿
        case *S: ❶
        case *R: ❷
        case S:  ❸
        case R:  ❹
        default: ❺
    }
}
```

⓿ 类型判断。在 `switch` 语句中使用 `(type)`。保存类型到变量 `t`；

❶ `p` 的实际类型是 `S` 的指针；

❷ `p` 的实际类型是 `R` 的指针；

❸ `p` 的实际类型是 `S`；

❹ `p` 的实际类型是 `R`；

❺ 实现了 `I` 的其他类型。

在 `switch` 之外使用 `(type)` 是非法的。类型判断不是唯一的*运行时*得到类型的方法。为了在*运行时*得到类型，同样可以使用 “comma, ok” 来判断一个接口类型是否实现了某个特定接口：

```
if t, ok := something.(I); ok {
    // 对于某些实现了接口 I 的
    // t 是其所拥有的类型
}
```
+

确定一个变量实现了某个接口，可以使用：

```
t := something.(I)
```

空接口

由于每个类型都能匹配到空接口：interface{}。我们可以创建一个接受空接口作为参数的普通函数：

Listing 5.2. 用空接口作为参数的函数

```
func g(something interface{}) int {
    return something.(I).Get()
}
```

在这个函数中的 return something.(I).Get() 是有一点窍门的。值 something 具有类型 interface{}，这意味着方法没有任何约束：它能包含任何类型。.(I) 是 类型断言，用于转换 something 到 I 类型的接口。如果有这个类型，则可以调用 Get() 函数。因此，如果创建一个 *S 类型的新变量，也可以调用 g()，因为 *S 同样实现了空接口。

```
s = new(S)
fmt.Println(g(s));
```

调用 g 的运行不会出问题，并且将打印 0。如果调用 g() 的参数没有实现 I 会带来一个麻烦：

Listing 5.3. 实现接口失败

```
i := 5                ← 声明 i 是一个 "该死的" int
fmt.Println(g(i))
```

这能编译，但是当运行的时候会得到：

```
panic: interface conversion: int is not main.I: missing method Get
```

这是绝对没问题，内建类型 int 没有 Get() 方法。

方法

方法就是有接收者的函数（参阅第 2 章）。

可以在任意类型上定义方法（除了非本地类型，包括内建类型：int 类型不能有方法）。然而可以新建一个拥有方法的整数类型。例如：

```
type Foo int

func (self Foo) Emit() {
  fmt.Printf("%v", self)
}

type Emitter interface {
  Emit()
}
```

对那些非本地（定义在其他包的）类型也一样：

Listing 5.4. 扩展内建类型错误

```
func (i int) Emit() {
  fmt.Printf("%d", i)
}
```

不能定义新的方法
在非本地类型 `int` 上

Listing 5.5. 扩展非本地类型错误

```
func (a *net.AddrError) Emit() {
  fmt.Printf("%v", a)
}
```

不能定义新的方法
在非本地类型 `net.AddrError` 上

接口类型的方法

接口定义为一个方法的集合。方法包含实际的代码。换句话说，一个接口就是定义，而方法就是实现。因此，接收者不能定义为接口类型，这样做的话会引起 `invalid receiver type ...` 的编译器错误。来自语言说明书 [10] 的权威内容：

> *接收者类型必须是 `T` 或 `*T`，这里的 `T` 是类型名。`T` 叫做接收者基础类型或简称基础类型。基础类型一定不能使指针或接口类型，并且定义在与方法相同的包中。*

> Pointers to interfaces
>
> 在 Go 中创建指向接口的指针是无意义的。实际上创建接口值的指针也是非法的。在 `2010-10-13` 的发布日志中进行的描述，使得没有任何余地怀疑这一事实：
>
> > *语言的改变是使用指针指向接口值不再自动反引用指针。指向接口值的指针通常是低级的错误，而不是正确的代码。*
>
> 这来自 [9]。如果不是这个限制，这个代码：
>
> ```
> var buf bytes.Buffer
> io.Copy(buf, os.Stdin)
> ```
>
> 就会复制标准输入到 `buf` 的副本，而不是 `buf` 本身。这看起来永远不会是一个期望的结果。

接口名字

根据规则，单方法接口命名为方法名加上 *-er* 后缀：Read*er*，Writ*er*，Formatt*er* 等。

有一堆这样的命名，高效的反映了它们职责和包含的函数名。`Read`，`Write`，`Close`，`Flush`，`String` 等等有着规范的声明和含义。为了避免混淆，除非有类似的声明和含义，否则不要让方法与这些重名。相反的，如果类型实现了与众所周知的类型相同的方法，那么就用相同的名字和声明；将字符串转换方法命名为 `String` 而不是 `ToString`。

文本复制于 [8]。

简短的例子

回顾那个冒泡排序的练习 Q13），对整型数组排序：

```
func bubblesort(n []int) {
    for i := 0; i < len(n)-1; i++ {
        for j := i + 1; j < len(n); j++ {
            if n[j] < n[i] {
                    n[i], n[j] = n[j], n[i]
            }
        }
    }
}
```

排序字符串的版本是类似的，除了函数的声明：

```
func bubblesortString(n []string) { /* ... */ }
```

基于此，可能会需要两个函数，每个类型一个。而通过使用接口可以让这个变得更加通用 。

来创建一个可以对字符串和整数进行排序的函数，这个例子的某些行是无法运行的：

```
func sort(i []interface{}) { ⓿
    switch i.(type) {         ❶
        case string:          ❷
            // ...
        case int:
            // ...
    }
    return /* ... */ ❸
}
```

⓿ 函数将接收一个空接口的 slice；

❶ 使用 type switch 找到输入参数实际的类型；

❷ 然后排序；

❸ 返回排序的 slice。

但是如果用 `sort([]int{1, 4, 5})` 调用这个函数，会失败：`cannot use i (type []int) as type []interface in function argument`

这是因为 Go 不能简单的将其转换为接口的 *slice*。转换到接口是容易的，但是转换到 slice 的开销就高了。

关于这个话题完整的邮件列表讨论可以在 [14] 这里找到。

简单来说 ：Go 不能（隐式）转换为 slice。

那么如何创建 Go 形式的这些 “通用” 函数呢？用 Go 隐式的处理来代替 type switch 方式的类型推断吧。下面的步骤是必须的：

1. 定义一个有着若干排序相关的方法的接口类型（这里叫做 `Sorter`）。至少需要获取 slice 长度的函数，比较两个值的函数和交换函数；

```
type Sorter interface {
    Len() int              ← len() 作为方法
    Less(i, j int) bool    ← p[j] < p[i] 作为方法
    Swap(i, j int)         ← p[i], p[j] = p[j], p[i] 作为方法
}
```

2. 定义用于排序 slice 的新类型。注意定义的是 slice 类型；

```
type Xi []int
type Xs []string
```

3. 实现 Sorter 接口的方法。整数的：

```
func (p Xi) Len() int               { return len(p) }
func (p Xi) Less(i int, j int) bool { return p[j] < p[i] }
func (p Xi) Swap(i int, j int)      { p[i], p[j] = p[j], p[i]
     }
```

和字符串的：

```
func (p Xs) Len() int               { return len(p) }
func (p Xs) Less(i int, j int) bool { return p[j] < p[i] }
func (p Xs) Swap(i int, j int)      { p[i], p[j] = p[j], p[i]
     }
```

4. 编写作用于 Sorter 接口的*通用*排序函数。

```
func Sort(x Sorter) { ⓿
    for i := 0; i < x.Len() - 1; i++ { ❶
        for j := i + 1; j < x.Len(); j++ {
            if x.Less(i, j) {
                x.Swap(i, j)
            }
        }
    }
}
```

⓿ x 现在是 Sorter 类型；

❶ 使用定义的函数，实现了冒泡排序。

现在可以像下面这样使用通用的 Sort 函数：

```
ints := Xi{44, 67, 3, 17, 89, 10, 73, 9, 14, 8}
strings := Xs{"nut", "ape", "elephant", "zoo", "go"}

Sort(ints)
fmt.Printf("%v\n", ints)
Sort(strings)
fmt.Printf("%v\n", strings)
```

在接口中列出接口

看一下下面的接口定义，这个是来自包 *container/heap* 的：

```
type Interface interface {
    sort.Interface
    Push(x interface{})
    Pop() interface{}
}
```

这里有另外一个接口在 `heap.Interface` 的定义中被列出，这看起来有些古怪，但是这的确是正确的，要记得接口只是一些方法的列表。`sort.Interface` 同样是这样一个列表，因此将其包含在接口内是毫无错误的。

自省和反射

在下面的例子中，了解一下定义在 `Person` 的定义中的 “标签”（这里命名为 “namestr”）。为了做到这个，需要 *reflect* 包（在 Go 中没有其他方法）。要记得，查看标签意味着返回*类型*的定义。因此使用 *reflect* 包来指出变量的类型，*然后*访问标签。

Listing 5.6. 使用反射自省

```
type Person struct {
    name string "namestr"      ← "namestr" 是标签
    age  int
}

func ShowTag(i interface{}) {         ← *Person 作为参数调用
 switch t := reflect.TypeOf(i); t.Kind() {
 case reflect.Ptr:                     ← 一个指针，也就是 reflect.Ptr
   tag := t.Elem().Field(0).Tag
```

⓿ We are dealing with a `Type` and according to the documentation[a]:

> *// Elem returns a type's element type.*
> *// It panics if the type's Kind is not Array, Chan, Map, Ptr, or Slice.*
> `Elem() Type`

同样的在 `t` 使用 `Elem()` 得到了指针指向的值。

❶ 现在已经定义了可以“深入”结构体的指针。

就可以使用 `Field(0)` 访问零值字段；

❷ 结构 `StructField` 有成员 `Tag`，返回字符串类型的标签名。因此，在第 0th 个字段上可以用 `.Tag` 访问这个名字：`Field(0).Tag`。这样得到了 `namestr`。

为了让类型和值之间的区别更加清晰，看下面的代码：

[a] `go doc reflect`

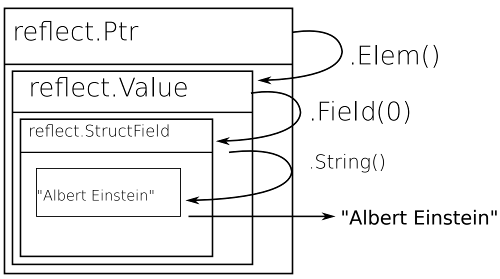
Listing 5.7. 反射类型和值

```
func show(i interface{}) {
    switch t := i.(type) {
      case *Person:
        t := reflect.TypeOf(i)       ← 得到类型的元数据
        v := reflect.ValueOf(i)      ← 得到实际的值
        tag := t.Elem().Field(0).Tag    ⓿
        name := v.Elem().Field(0).String() ❶
    }
}
```

⓿ 这里希望获得 "标签"。因此需要 `Elem()` 重定向至其上，访问第一个字段来获取标签。注意将 `t` 作为一个 `reflect.Type` 来操作；

❶ 现在需要访问其中一个成员的*值*，并让 `v` 上的 `Elem()` 进行重定向。这样就访问到了结构。然后访问第一个字段 `Field(0)` 并且调用其上的 `String()` 方法。

Figure 5.1. 用反射去除层次关系。通过 `Elem()` *访问* `*Person`*，使用* `go doc reflect` *中描述的方法获得* `string` *内部包含的内容。*



设置值与获得值类似，但是仅仅工作在*可导出*的成员上。这些代码：

Listing 5.8. 私有成员的反射

```
type Person struct {
 name string      ← 名称
 age  int
}

func Set(i interface{}) {
 switch i.(type) {
 case *Person:
  r := reflect.ValueOf(i)
  r.Elem(0).Field(0).SetString("
      Albert Einstein")
  }
}
```

Listing 5.9. 公有成员的反射

```
type Person struct {
 Name string      ← Name
 age  int
}

func Set(i interface{}) {
 switch i.(type) {
 case *Person:
  r := reflect.ValueOf(i)
  r.Elem().Field(0).SetString("
      Albert Einstein")
  }
}
```

左边的代码可以编译并运行，但是当运行的时候，将得到打印了栈的*运行时*错误：

`panic: reflect.Value.SetString using value obtained using unexported field`

右边的代码没有问题，并且设置了成员变量 `Name` 为 “Albert Einstein”。当然，这仅仅工作于调用 `Set()` 时传递一个指针参数。

练习

Q23. (1) 接口和编译

1. 在第 72 页的代码 5.3 编译正常——就像文中开始描述的那样。但是当运行的时候，会得到运行时错误，因此有些东西有错误。为什么代码编译没有问题呢？

Q24. (1) 指针和反射

1. 在第 “自省和反射” 节，第 76 页的最后一段中，有这样的描述：

   > *右边的代码没有问题，并且设置了成员变量* `Name` *为 “Albert Einstein”。当然，这仅仅工作于调用* `Set()` *时传递一个指针参数。*

   为什么是这样的情况？

Q25. (2) 接口和 max()

1. 在练习 Q12 中创建了工作于一个整形 slice 上的最大函数。现在的问题是创建一个显示最大数字的程序，同时工作于整数和浮点数。虽然在这里会相当困难，不过还是让程序尽可能的通用吧。

答案

A23. (1) 接口和编译

1. 代码能够编译是因为整数类型实现了空接口，这是在编译时检查的。

   修复这个正确的途径是测试这个空接口可以被转换，如果可以，调用对应的方法。5.2 列出的 Go 代码中定义了函数 g——这里重复一下：

   ```
   func g(any interface{}) int { return any.(I).Get() }
   ```

   应当修改为：

   ```
   func g(any interface{}) int {
       if v, ok := any.(I); ok {   // 检查是否可以转换
           return v.Get()          // 如果可以，调用 Get()
       }
       return -1                   // 随便返回个什么
   }
   ```

   如果现在调用 g()，就不会有运行时错误了。在 Go 中这种用法被称作 "comma ok"。

A24. (1) 指针和反射

1. 当调用一个非指针参数，变量是复制（call-by-value）的。因此，进行魔法般的反射是在副本上。这样就*不能*改变原来的值，仅仅改变副本。

A25. (2) 接口和 max()

1. 下面的程序计算了最大值。它是 Go 能做到的最通用的形式了。

   Listing 5.10. 通用的计算最大值

   ```
   package main

   func Less(l, r interface{}) bool { ⓿
           switch l.(type) {
           case int:
                   if _, ok := r.(int); ok {
                           return l.(int) < r.(int) ❶
                   }
           case float32:
                   if _, ok := r.(float32); ok {
                           return l.(float32) < r.(float32) ❷
                   }
           }
           return false
   }

   func main() {
   ```

```
        var a, b, c int = 5, 15, 0
        var x, y, z float32 = 5.4, 29.3, 0.0

       if c = a; Less(a, b) { ❸
               c = b
        }
       if z = x; Less(x, y) { ❹
               z = y
        }
        println(c, z)
}
```

⓿ 也可以选择让这个函数的返回值为 `interface{}`，但是这也就意味着调用者不得不总是使用类型断言来从接口中解析出实际的类型；

❶ 所有类型定义为整数。然后进行比较；

❷ 参数是 `float32`；

❸ 获得 a 和 b 中的最大值；

❹ 浮点类型也一样。

6 并发

- “并行是关于性能的；
- 并发是关于程序设计的。”

Google IO 2010
ROB PIKE

在这章中将展示 Go 使用 channel 和 goroutine 开发并行程序的能力。goroutine 是 Go 并发能力的核心要素。但是，goroutine 到底 是什么？来自 [8]：

> *叫做 goroutine 是因为已有的短语——线程、协程、进程等等——传递了不准确的含义。goroutine 有简单的模型：*它是与其他 goroutine 并行执行的，有着相同地址空间的函数。*。它是轻量的，仅比分配栈空间多一点点消耗。而初始时栈是很小的，所以它们也是廉价的，并且随着需要在堆空间上分配（和释放）。*

goroutine 是一个普通的函数，只是需要使用关键字 go 作为开头。

```
ready("Tea", 2)          ← 普通函数调用
go ready("Tea", 2)       ← ready() 作为 goroutine 运行
```

下面程序的思路来自 [20]。让一个函数作为两个 goroutine 执行，goroutine 等待一段时间，然后打印一些内容到屏幕。在第 14 和 15 行，启动了 goroutine。`main` 函数等待足够的长的时间，这样每个 goroutine 会打印各自的文本到屏幕。现在是在第 17 行等待 5 秒钟，但实际上没有任何办法知道，当所有 goroutine 都已经退出应当等待多久。

Listing 6.1. Go routine 实践

```
8  func ready(w string, sec int) {
9          time.Sleep(time.Duration(sec) * time.Second)
10         fmt.Println(w, "is ready!")
11 }

13 func main() {
14         go ready("Tea", 2)
15         go ready("Coffee", 1)
16         fmt.Println("I'm waiting")
17         time.Sleep(5 * time.Second)
18 }
```

表 6.1 输出：

```
I'm waiting          ← 立刻
Coffee is ready!     ← 1 秒后
Tea is ready!        ← 2 秒后
```

如果不等待 goroutine 的执行（例如，移除第 17 行），程序立刻终止，而任何正在执行的 goroutine 都会*停止*。为了修复这个，需要一些能够同 goroutine 通讯的机制。这一机制通过 channels 的形式使用。channel 可以与 Unix sehll 中的双向管道做类比：可以通过它发送或者接收值。这些值只能是特定的类型：channel 类型。定义一个 channel 时，也需要定义发送到 channel 的值的类型。注意，必须使用 `make` 创建 channel：

```
ci := make(chan int)
cs := make(chan string)
cf := make(chan interface{})
```

创建 channel `ci` 用于发送和接收整数，创建 channel `cs` 用于字符串，以及 channel `cf` 使用了空接口来满足各种类型。向 channel 发送或接收数据，是通过类似的操作符完成的：`<−`. 具体作用则依赖于操作符的位置：

```
ci <− 1        ← 发送整数 1 到 channel ci
<−ci           ← 从 channel ci 接收整数
i := <−ci      ← 从 channel ci 接收整数，并保存到 i 中
```

将这些放到实例中去。

Listing 6.2. Go routines 和 channel

```
var c chan int ⓿

func ready(w string, sec int) {
        time.Sleep(time.Duration(sec) * time.Second)
        fmt.Println(w, "is ready!")
        c <− 1  ❶
}

func main() {
        c = make(chan int) ❷
        go ready("Tea", 2) ❸
        go ready("Coffee", 1)
        fmt.Println("I'm waiting, but not too long")
        <−c ❹
        <−c ❺
}
```

⓿ 定义 `c` 作为 int 型的 channel。就是说：这个 channel 传输整数。注意这个变量是全局的，这样 goroutine 可以访问它；

❶ 发送整数 1 到 channel `c`；

❷ 初始化 `c`；

❸ 用关键字 go 开始一个 goroutine；

❹ 等待，直到从 channel 上接收一个值。注意，收到的值被丢弃了；

❺ 两个 goroutines，接收两个值。

这里仍然有一些丑陋的东西；不得不从 channel 中读取两次（第 14 和 15 行）。在这个例子中没问题，但是如果不知道有启动了多少个 goroutine 怎么办呢？这里有另一个 Go 内建的关键字：`select`。通过 `select`（和其他东西）可以监听 channel 上输入的数据。

在这个程序中使用 `select`，并不会让它变得更短，因为运行的 goroutine 太少了。移除第 14 和 15 行，并用下面的内容替换它们：

Listing 6.3. 使用 select

```
14 L: for {
15         select {
16         case <-c:
17                 i++
18                 if i > 1 {
19                         break L
20                 }
21         }
22 }
```

现在将会一直等待下去。只有当从 channel `c` 上收到多个响应时才会退出循环 `L`。

使其并行运行

虽然 goroutine 是并发执行的，但是它们并不是并行运行的。如果不告诉 Go 额外的东西，同一时刻只会有一个 goroutine 执行。利用 `runtime.GOMAXPROCS(n)` 可以设置 goroutine 并行执行的数量。来自文档：

> *GOMAXPROCS 设置了同时运行的 CPU 的最大数量，并返回之前的设置。如果 n < 1，不会改变当前设置。*当调度得到改进后，这将被移除。

如果不希望修改任何源代码，同样可以通过设置环境变量 `GOMAXPROCS` 为目标值。

更多关于 channel

当在 Go 中用 `ch := make(chan bool)` 创建 chennel 时，bool 型的 无缓冲 channel 会被创建。这对于程序来说意味着什么呢？首先，如果读取（`value := <-ch`）它将会被阻塞，直到有数据接收。其次，任何发送（`ch<-5`）将会被阻塞，直到数据被读出。无缓冲 channel 是在多个 goroutine 之间同步很棒的工具。

不过 Go 也允许指定 channel 的缓冲大小，很简单，就是 channel 可以存储多少元素。`ch := make(chan bool, 4)`，创建了可以存储 4 个元素的 bool 型 channel。在这个 channel 中，前 4 个元素可以无阻塞的写入。当写入第 5[⊠] 元素时，代码 *将会*阻塞，直到其他 goroutine 从 channel 中读取一些元素，腾出空间。

一句话说，在 Go 中下面的为 true：

$$\texttt{ch := make(chan type, value)} \begin{cases} value == 0 & \rightarrow \text{无缓冲} \\ value > 0 & \rightarrow \text{缓冲 } \textit{value} \text{ 的元素} \end{cases}$$

关闭 channel

当 channel 被关闭后，读取端需要知道这个事情。下面的代码演示了如何检查 channel 是否被关系。

```
x, ok = <-ch
```

当 `ok` 被赋值为 `true` 意味着 channel 尚未被关闭，*同时* 可以读取数据。否则 `ok` 被赋值为 `false`。在这个情况下表示 channel 被关闭。

只读或只写 channel

练习

Q26. (1) Channel

1. 修改在练习 Q1 中创建的程序，换句话说，主体中调用的函数现在是一个 goroutine 并且使用 channel 通讯。不用担心 goroutine 是如何停止的。
2. 在完成了问题 1 后，仍有一些待解决的问题。其中一个麻烦是 goroutine 在 `main.main()` 结束的时候，没有进行清理。更糟的是，由于 `main.main()` 和 `main.shower()` 的竞争关系，不是所有数字都被打印了。本应该打印到 9，但是有时只打印到 8。添加第二个退出 channel，可以解决这两个问题。试试吧。[a]

Q27. (2) 斐波那契 II

1. 这是类似的练习，第一个在第 34 页的练习 10。完整的问题描述：

   斐波那契数列以： $1, 1, 2, 3, 5, 8, 13, \ldots$ *开头。或用数学形式：* $x_1 = 1; x_2 = 1; x_n = x_{n-1} + x_{n-2} \quad \forall n > 2$。
   编写一个函数接收 `int` *值，并给出同样数量的斐波那契数列。*

   *但是*现在有额外条件：必须使用 channel。

[a]需要用到 `select` 语句。

答案

A26. (1) Channel

1. 程序可能的形式是：

Listing 6.4. Go 的 channel

```
1 package main

3 import "fmt"

5 func main() {
6         ch := make(chan int)
7         go shower(ch)
8         for i := 0; i < 10; i++ {
9                 ch <- i
10        }
11 }

13 func shower(c chan int) {
14        for {
15                j := <-c
16                fmt.Printf("%d\n", j)
17        }
18 }
```

以通常的方式开始，在第 6 行创建了一个新的 int 类型的 channel。下一行调用了 shower 函数，用 ch 变量作为参数，这样就可以与其通讯。然后进入 for 循环（第 8-10 行），在循环中发送（通过 <-）数字到函数（现在是 goroutine）shower。在函数 shower 中等待（阻塞方式），直到接收到了数字（第 15 行）。每个收到的数字都被打印（第 16 行）出来，然后继续第 14 行开始的死循环。

2. 答案是

Listing 6.5. 添加额外的退出 channel

```
package main

import "fmt"

func main() {
        ch := make(chan int)
        quit := make(chan bool)
        go shower(ch, quit)
        for i := 0; i < 10; i++ {
                ch <- i
        }
        quit <- false   // 或者是 true，这没啥关系
}
```

```
15 func shower(c chan int, quit chan bool) {
16         for {
17                 select {
18                 case j := <-c:
19                         fmt.Printf("%d\n", j)
20                 case <-quit:
21                         break
22                 }
23         }
24 }
```

在第 20 行从退出 channel 读取并丢弃该值。可以使用 `q := <-quit`，但是可能只需要用这个变量一次——在 Go 中是非法的。另一种办法，你可能已经想到了：`_ = <-quit`。在 Go 中这是合法的，但是第 20 行的形式在 Go 中更好。

A27. (2) 斐波那契 II

1. 下面的程序使用 channel 计算了斐波那契数列。

Listing 6.6. Go 的斐波那契函数

```
package main
import "fmt"

func dup3(in <-chan int) (<-chan int, <-chan int, <-chan int
    ) {
        a, b, c := make(chan int, 2), make(chan int, 2),
            make(chan int, 2)
        go func() {
                for {
                        x := <-in
                        a <- x
                        b <- x
                        c <- x
                }
        }()
        return a, b, c
}

func fib() <-chan int {
        x := make(chan int, 2)
        a, b, out := dup3(x)
        go func() {
                x <- 0
                x <- 1
                <-a
                for {
```

```
                        x <- <-a+<-b
                }
        }()
        return out
}

func main() {
        x := fib()
        for i := 0; i < 10; i++ {
                fmt.Println(<-x)
        }
}

// See sdh33b.blogspot.com/2009/12/fibonacci-in-go.html
```

7 通讯

> “好的沟通就像是一杯刺激的浓咖啡，然后就难以入睡。”
>
> ANNE MORROW LINDBERGH

在这章中将介绍 Go 中与外部通讯的通讯模块。将会了解文件、目录、网络通讯和运行其他程序。Go 的 I/O 核心是接口 `io.Reader` 和 `io.Writer`。

在 Go 中，从文件读取（或写入）是非常容易的。程序只需要使用 *os* 包就可以从文件 `/etc/passwd` 中读取数据。

Listing 7.1. 从文件读取（无缓冲）

```
package main

import "os"

func main() {
        buf := make([]byte, 1024)
        f, _ := os.Open("/etc/passwd") ⓿
        defer f.Close() ❶
        for {
                n, _ := f.Read(buf) ❷
                if n == 0 { break } ❸
                os.Stdout.Write(buf[:n]) ❹
        }
}
```

接下来展示了如何做到这点：

⓿ 打开文件，`os.Open` 返回一个实现了 `io.Reader` 和 `io.Writer` 的 `*os.File`；

❶ 确保关闭了 `f`；

❷ 一次读取 1024 字节；

❸ 到达文件末尾；

❹ 将内容写入 `os.Stdout`

如果想要使用缓冲 IO，则有 *bufio* 包：

Listing 7.2. 从文件读取（缓冲）

```
package main

import ( "os"; "bufio")

func main() {
```

```
        buf := make([]byte, 1024)
        f, _ := os.Open("/etc/passwd") ⓿
        defer f.Close()
        r := bufio.NewReader(f) ❶       w := bufio.NewWriter(os.
            Stdout)
        defer w.Flush()
        for {
                n, _ := r.Read(buf)     ❷
                if n == 0 { break }
                w.Write(buf[0:n])
        }
}
```

⓿ 打开文件；

❶ 转换 `f` 为有缓冲的 `Reader`。`NewReader` 需要一个 `io.Reader`，因此或许你认为这会出错。但其实不会。*任何*有 `Read()` 函数就实现了这个接口。同时，从列表7.1可以看到，`*os.File` 已经这样做了；

❷ 从 Reader 读取，而向 Writer 写入，然后向屏幕输出文件。

io.Reader

在前面已经提到 io.Reader 接口对于 Go 语言来说非常重要。许多（如果不是全部的话）函数需要通过 `io.Reader` 读取一些数据作为输入。为了满足这个接口，只需要实现一个方法：`Read(p []byte) (n int, err error)`。写入则是（你可能已经猜到了）实现了 `Write` 方法的 `io.Writer`。如果你让自己的程序或者包中的类型实现了 `io.Reader` 或者 `io.Writer` 接口，那么*整个 Go 标准库都可以使用*这个类型！

一些例子

前面的程序将整个文件读出，但是通常情况下会希望一行一行的读取。下面的片段展示了如何实现：

```
f, _ := os.Open("/etc/passwd"); defer f.Close()
r := bufio.NewReader(f)      ← 使其成为一个 bufio，以便访问 ReadString 方法
s, ok := r.ReadString('\n') {     ← 从输入中读取一行
// ... |     ← s 保存了字符串，通过 string 包就可以解析它|
```

更加通用（但是也更加复杂）的方法是 `ReadLine`，参阅包 *bufio* 的文档了解更多内容。

在 shell 脚本中通常遇到的场景是需要检查某个目录是否存在，如果不存在，就创建一个。

Listing 7.3. 用 shell 创建一个目录

```
if [ ! -e name ]; then
    mkdir name
else
    # error
fi
```

Listing 7.4. 用 Go 创建一个目录

```
if f, e := os.Stat("name"); e !=
     nil {
    os.Mkdir("name", 0755)
} else {
    // error
}
```

这两个例子的相似之处展示了 Go 拥有的“脚本”化特性，例如，用 Go 编写程序感觉上类似使用动态语言（Python、Ruby、Perl 或者 PHP）。

命令行参数

来自命令行的参数在程序中通过字符串 slice `os.Args` 获取，导入包 *os* 即可。*flag* 包有着精巧的接口，同样提供了解析标识的方法。这个例子是一个 DNS 查询工具：

```
dnssec := flag.Bool("dnssec", false, "Request DNSSEC records") ⓿
port := flag.String("port", "53", "Set the query port")       ❶
flag.Usage = func() {   ❷
    fmt.Fprintf(os.Stderr, "Usage: %s [OPTIONS] [name ...]\n", os.
        Args[0])
    flag.PrintDefaults() ❸
}
flag.Parse()    ❹
```

⓿ 定义 `bool` 标识，`-dnssec`。变量必须是指针，否则 package 无法设置其值；

❶ 类似的，`port` 选项；

❷ 简单的重定义 `Usage` 函数，有点罗嗦；

❸ 指定的每个标识，`PrintDefaults` 将输出帮助信息；

❹ 解析标识，并填充变量。

当参数被解析之后，就可以使用它们：

```
if *dnssec {          ← 定义传入参数 dnssec
    // 做点啥
}
```

执行命令

os/exec 包有函数可以执行外部命令，这也是在 Go 中主要的执行命令的方法。通过定义一个有着数个方法的 `*exec.Cmd` 结构来使用。

执行 `ls -l`：

```
import "os/exec"
```

```
cmd := exec.Command("/bin/ls", "-l")
err := cmd.Run()
```

上面的例子运行了 “ls -l”，但是没有对其返回的数据进行任何处理，通过如下方法从命令行的标准输出中获得信息：

```
import "exec"

cmd := exec.Command("/bin/ls", "-l")
buf, err := cmd.Output()         ← buf 是一个 []byte
```

网络

所有网络相关的类型和函数可以在 *net* 包中找到。这其中最重要的函数是 `Dial`。当 `Dial` 到远程系统，这个函数返回 `Conn` 接口类型，可以用于发送或接收信息。函数 `Dial` 简洁的抽象了网络层和传输层。因此 IPv4 或者 IPv6，TCP 或者 UDP 可以共用一个接口。

通过 TCP 连接到远程系统（端口 80），然后是 UDP，最后是 TCP 通过 IPv6，大致是这样 [a]：

```
conn, e := Dial("tcp", "192.0.32.10:80")
conn, e := Dial("udp", "192.0.32.10:80")
conn, e := Dial("tcp", "[2620:0:2d0:200::10]:80")
   ← 方括号是强制的
```

如果没有错误（由 e 返回），就可以使用 conn 从套接字中读写。在包 *net* 中的原始定义是：

> // Read *reads data from the connection.*
> Read(b []byte)(n int, err error)

这使得 conn 成为了 `io.Reader`。

> // Write *writes data to the connection.*
> Write(b []byte)(n int, err error)

这同样使得 conn 成为了 `io.Writer`，事实上 conn 是 `io.ReadWriter`。[b]

但是这些都是隐含的低层[c]，通常总是应该使用更高层次的包。例如 *http* 包。一个简单的 http Get 作为例子：

```
package main
import ( "io/ioutil"; "net/http"; "fmt" ) ⓿

func main() {
    r, err := http.Get("http://www.google.com/robots.txt") ❶
    if err != nil { fmt.Printf("%s\n", err.String()); return } ❷
```

[a] 在这个例子中，可以认为 192.0.32.10 和 2620:0:2d0:200::10 是 `www.example.org`。
[b] 变量 conn 同样实现了 `close` 方法，这使其成为一个 `io.ReadWriteCloser`。
[c] 练习 Q32 是关于使用这些的。

```
    b, err := ioutil.ReadAll(r.Body)      ❸
    r.Body.Close()
    if err == nil { fmt.Printf("%s", string(b)) } ❹
}
```

⓿ 需要的导入；

❶ 使用 http 的 Get 获取 html；

❷ 错误处理；

❸ 将整个内容读入 b；

❹ 如果一切 OK 的话，打印内容。

练习

Q28. (2) 进程

1. 编写一个程序，列出所有正在运行的进程，并打印每个进程执行的子进程个数。输出应当类似：

   ```
   Pid 0 has 2 children: [1 2]
   Pid 490 has 2 children: [1199 26524]
   Pid 1824 has 1 child: [7293]
   ```

   - 为了获取进程列表，需要得到 ps -e -opid,ppid,comm 的输出。输出类似：

     ```
       PID  PPID COMMAND
      9024  9023 zsh
     19560  9024 ps
     ```

   - 如果父进程有一个子进程，就打印 child，如果多于一个，就打印 children；
   - 进程列表要按照数字排序，这样就以 pid 0 开始，依次展示。

   这里有一个 Perl 版本的程序来帮助上手（或者造成绝对的混乱）。

   Listing 7.5. Perl 显示进程

   ```
   #!/usr/bin/perl -l
   my (%child, $pid, $parent);
   my @ps=`ps -e -opid,ppid,comm`;     # Capture the output
       from `ps`
   foreach (@ps[1..$#ps]) {            # Discard the header
       line
       ($pid, $parent, undef) = split; # Split the line,
           discard 'comm'
       push @{$child{$parent}}, $pid;  # Save the child PIDs on
            a list
   ```

```
}
# Walk through the sorted PPIDs
foreach (sort { $a <=> $b } keys %child) {
    print "Pid ", $_, " has ", @{$child{$_}}+0, " child",
        @{$child{$_}} == 1 ? ": " : "ren: ", "[@{$child{$_}}
            ]";
}
```

Q29. (0) 单词和字母统计

1. 编写一个从标准输入中读取文本的小程序，并进行下面的操作：

    1. 计算字符数量（包括空格）；
    2. 计算单词数量；
    3. 计算行数。

    换句话说，实现一个 `wc(1)`（参阅本地的手册页面），然而只需要从标准输入读取。

Q30. (0) Uniq

1. 编写一个 Go 程序模仿 Unix 命令 `uniq` 的功能。程序应当像下面这样运行，提供一个下面这样的列表：

    `'a' 'b' 'a' 'a' 'a' 'c' 'd' 'e' 'f' 'g'`

    它将打印出没有后续重复的项目：

    `'a' 'b' 'a' 'c' 'd' 'e' 'f'`

    下面列出的 7.8 是 Perl 实现的算法。

    Listing 7.8. uniq(1) 的 Perl 实现

```
#!/usr/bin/perl
my @a = qw/a b a a a c d e f g/;
print my $first = shift @a;
foreach (@a) {
    if ($first ne $_) { print; $first = $_; }
}
```

Q31. (2) Quine *Quine* 是一个打印自己的程序。

1. 用 Go 编写一个 Quine 程序。

Q32. (1) Echo 服务

1. 编写一个简单的 echo 服务。使其监听于本地的 TCP 端口 8053 上。它应当可以读取一行（以换行符结尾），将这行原样返回然后关闭连接。
2. 让这个服务可以并发，这样每个请求都可以在独立的 goroutine 中进行处理。

Q33. (2) 数字游戏

- 从列表中随机选择六个数字：

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 25, 50, 75, 100$$

  数字可以多次被选中；

- 从 $1 \ldots 1000$ 中选择一个随机数 i；

- 尝试用先前的六个数字（或者其中的几个）配合运算符 $+$，$-$，$*$ 和 $/$，计算出 i；

例如，选择了数字：1，6，7，8，8 和 75。并且 i 为 977。可以用许多方法来实现，其中一种：

$$((((1 * 6) * 8) + 75) * 8) - 7 = 977$$

或者

$$(8 * (75 + (8 * 6))) - (7/1) = 977$$

1. 实现像这样的数字游戏。使其打印像上面那样格式的结果（也就是说，输出应当是带有括号的中序表达式）

2. 计算全*部*可能解，并且全部显示出来（或者仅显示有多少个）。在上面的例子中，有 544 种方法。

Q34. (1) *Finger 守护进程

1. 编写一个 finger 守护进程，可以工作于 finger(1) 命令。

   来自 Debian 的包描述：

   > *Fingerd 是一个基于 RFC 1196 [28] 的简单的守护进程，它为许多站点提供了 "finger" 程序的接口。这个程序支持返回一个友好的、面向用户的系统或用户当前状况的详细报告。*

   最基本的只需要支持用户名参数。如果用户有 `.plan` 文件，则显示该文件内容。因此程序需要能够提供：

   - 用户存在吗？
   - 如果用户存在，显示 `.plan` 文件的内容。

答案

A28. (2) 进程

1. 有许多工作需要做。可以将程序分为以下几个部分：

    1. 运行 ps 获得输出；
    2. 解析输出并保存每个 PPID 的子 PID；
    3. 排序 PPID 列表；
    4. 打印排序后的列表到屏幕。

    在下面的解法中，使用了一个 `map[int][]int`，就是一个使用整数作为 map 的索引，元素是整数的 slice ——用于保存 PID。内建的 append 被用于扩展这个整数的 slice。

    程序清单：

    Listing 7.6. Go 显示进程

```
package main

import ( "fmt"; "os/exec"; "sort"; "strconv"; "strings")

func main() {
        ps := exec.Command("ps", "-e", "-opid,ppid,comm")
        output, _ := ps.Output()
        child := make(map[int][]int)
        for i, s := range strings.Split(string(output), "\n"
            ) {
                if i == 0 { continue } // Kill first line
                if len(s) == 0 { continue } // Kill last line
                f := strings.Fields(s)
                fpp, _ := strconv.Atoi(f[1]) // 父 pid
                fp, _ := strconv.Atoi(f[0])  // 子 pid
                child[fpp] = append(child[fpp], fp)
        }
        schild := make([]int, len(child))
        i := 0
        for k, _ := range child { schild[i] = k; i++ }
        sort.Ints(schild)
        for _, ppid := range schild {
                fmt.Printf("Pid %d has %d child", ppid, len(
                    child[ppid]))
                if len(child[ppid]) == 1 {
                        fmt.Printf(": %v\n", child[ppid])
                        continue
                }
                fmt.Printf("ren: %v\n", child[ppid])
```

```
        }
}
```

A29. (0) 单词和字母统计

1. 下面是 wc(1) 的一种实现。

Listing 7.7. wc(1) 的 Go 实现

```
package main

import (
        "os"
        "fmt"
        "bufio"
        "strings"
)

func main() {
        var chars, words, lines int
        r := bufio.NewReader(os.Stdin) ⓿
        for {
                switch s, ok := r.ReadString('\n'); true { ❶
                case ok != nil: ❷
                        fmt.Printf("%d %d %d\n", chars,
                            words, lines);
                        return
                default:        ❸
                        chars += len(s)
                        words += len(strings.Fields(s))
                        lines++
                }
        }
}
```

⓿ Start a new reader that reads from standard input;

❶ Read a line from the input;

❷ If we received an error, we assume it was because of a EOF. So we print the current values;

❸ Otherwise we count the charaters, words and increment the lines.

A30. (0) Uniq

1. 下面是 uniq 的 Go 实现.

Listing 7.9. uniq(1) 的 Go 实现

```
package main
```

```
import "fmt"

func main() {
        list := []string{"a", "b", "a", "a", "c", "d", "e",
            "f"}
        first := list[0]

        fmt.Printf("%s ", first)
        for _, v := range list[1:] {
                if first != v {
                        fmt.Printf("%s ", v)
                        first = v
                }
        }

}
```

A31. (2) Quine

下面是来自 Russ Cox 提交在 Go Nuts 邮件列表上的解决方案。

Listing 7.10. 一个 Go quine

1.
```
/* Go quine */
package main
import "fmt"
func main() {
 fmt.Printf("%s%c%s%c\n", q, 0x60, q, 0x60)
}
var q = `/* Go quine */
package main
import "fmt"
func main() {
 fmt.Printf("%s%c%s%c\n", q, 0x60, q, 0x60)
}
var q = `
```

A32. (1) Echo 服务

1. 一个简单的 echo 服务器是这样：

Listing 7.11. 简易 echo 服务器

```
package main
import ( "net"; "fmt";"bufio" )
```

```
func main() {
        l, err := net.Listen("tcp", "127.0.0.1:8053")
        if err != nil {
                fmt.Printf("Failure to listen: %s\n", err.
                    Error())
        }
        for {
                if c, err := l.Accept(); err == nil { Echo(c
                    ) }
        }
}

func Echo(c net.Conn) {
        defer c.Close()
        line, err := bufio.NewReader(c).ReadString('\n')
        if err != nil {
                fmt.Printf("Failure to read: %s\n", err.
                    Error())
                return
        }
        _, err = c.Write([]byte(line))
        if err != nil {
                fmt.Printf("Failure to write: %s\n", err.
                    Error())
                return
        }
}
```

当运行起来的时候可以看到如下内容：

```
% nc 127.0.0.1 8053
Go is *awesome*
Go is *awesome*
```

2. 为了使其能够并发处理链接，*只需要修改一行代码*，就是：

   ```
   if c, err := l.Accept(); err == nil { Echo(c) }
   ```

   改为：

   ```
   if c, err := l.Accept(); err == nil { go Echo(c) }
   ```

A33. (2) 数字游戏

1. 下面的是一种可能的解法。它使用了递归和回溯来得到答案。

Listing 7.12. 数字游戏

```
package main

import ( "fmt"; "strconv"; "flag")
```

```
const (
        _ = 1000 * iota
        ADD
        SUB
        MUL
        DIV
        MAXPOS = 11
)

var mop = map[int]string{ADD: "+", SUB: "-", MUL: "*", DIV:
    "/"}

var (
        ok    bool
        value int
)

type Stack struct {
        i    int
        data [MAXPOS]int
}

func (s *Stack) Reset()     { s.i = 0 }

func (s *Stack) Len() int   { return s.i }

func (s *Stack) Push(k int) { s.data[s.i] = k; s.i++ }

func (s *Stack) Pop() int   { s.i--; return s.data[s.i] }

var found int
var stack = new(Stack)

func main() {
        flag.Parse()
        list := []int{1, 6, 7, 8, 8, 75, ADD, SUB, MUL, DIV}
        magic, ok := strconv.Atoi(flag.Arg(0))  // Arg0 是 i
        if ok != nil { return }
        f := make([]int, MAXPOS)
        solve(f, list, 0, magic)
}

func solve(form, numberop []int, index, magic int) {
        var tmp int
```

```
        for i, v := range numberop {
                if v == 0 { goto NEXT }
                if v < ADD { // 是一个数字，保存起来
                        tmp = numberop[i]
                        numberop[i] = 0
                }
                form[index] = v
                value, ok = rpncalc(form[0 : index+1])

                if ok && value == magic {
                        if v < ADD {
                                numberop[i] = tmp // 重置并继
                                    续
                        }
                        found++
                        fmt.Printf("%s = %d  #%d\n", rpnstr(
                            form[0:index+1]), value, found)
                }

                if index == MAXPOS-1 {
                        if v < ADD {
                                numberop[i] = tmp // 重置并继
                                    续
                        }
                        goto NEXT
                }
                solve(form, numberop, index+1, magic)
                if v < ADD {
                        numberop[i] = tmp // 重置并继续
                }
        NEXT:
        }
}

func rpnstr(r []int) (ret string) { // 将 rpn 转换到固定的标记
        s := make([]string , 0) // 分配内存
        for k, t := range r {
                switch t {
                case ADD, SUB, MUL, DIV:
                        a, s := s[len(s)-1], s[:len(s)-1]
                        b, s := s[len(s)-1], s[:len(s)-1]
                        if k == len(r)-1 {
                                s = append(s, b+mop[t]+a)
                        } else {
                                s = append(s, "("+b+mop[t]+a
```

```
                                        +")")
                                }
                        default:
                                s = append(s, strconv.Itoa(t))
                        }
                }
                for _, v := range s { ret += v }
                return
        }

        func rpncalc(r []int) (int, bool) {
                stack.Reset()
                for _, t := range r {
                        switch t {
                        case ADD, SUB, MUL, DIV:
                                if stack.Len() < 2 { return 0, false
                                    }
                                a := stack.Pop()
                                b := stack.Pop()
                                if t == ADD { stack.Push(b + a) }
                                if t == SUB {
                                        // 不接受负数
                                        if b-a < 0 {
                                                return 0, false
                                        }
                                        stack.Push(b - a)
                                }
                                if t == MUL { stack.Push(b * a) }
                                if t == DIV {
                                        if a == 0 {
                                                return 0, false
                                        }
                                        // 不接受余数
                                        if b%a != 0 {
                                                return 0, false
                                        }
                                        stack.Push(b / a)
                                }
                        default:
                                stack.Push(t)
                        }
                }
                if stack.Len() == 1 { // 只有一个！
                        return stack.Pop(), true
                }
```

```
        return 0, false
}
```

2. 开始运行 permrec 时，输入 977 作为第一个参数：

```
% ./permrec 977
1+(((6+7)*75)+(8/8)) = 977  #1
...                         ...
((75+(8*6))*8)-7 = 977      #542
(((75+(8*6))*8)-7)*1 = 977  #543
(((75+(8*6))*8)-7)/1 = 977  #544
```

A34. (1) *Finger 守护进程

| 这是来自 Fabian Becker 的解决方案。

Listing 7.13. finger 守护进程

1.
```
package main

import (
        "bufio"
        "errors"
        "flag"
        "io/ioutil"
        "net"
        "os/user"
        "strconv"
)

func main() {
        flag.Parse()
        ln, err := net.Listen("tcp", ":79")
        if err != nil {
                panic(err)
        }
        for {
                conn, err := ln.Accept()
                if err != nil {
                        continue
                }
                go handleConnection(conn)
        }
}

func handleConnection(conn net.Conn) {
        defer conn.Close()
        reader := bufio.NewReader(conn)
        usr, _, _ := reader.ReadLine()
```

```
        if info, err := getUserInfo(string(usr)); err != nil
            {
                conn.Write([]byte(err.Error()))
        } else {
                conn.Write(info)
        }
}

func getUserInfo(usr string) ([]byte, error) {
        u, e := user.Lookup(usr)
        if e != nil {
                return nil, e
        }
        data, err := ioutil.ReadFile(u.HomeDir + ".plan")
        if err != nil {
                return data, errors.New("User doesn't have a
                    .plan file!\n")
        }
        return data, nil
}
```

A 版权

本作品由 LaTeX 创作。主文本设置为 Google Droid 字体。所有打印文本设置为 DejaVu Mono。中文字体设置为 UKai。

贡献者

下面的成员编写了这本书。

- Miek Gieben `<miek@miek.nl>`；
- JC van Winkel；
- 邢星，中文译本。

帮助试读、检查练习和改进文案（不论真实姓名还是别名都不分排名先后）*Adam J. Gray*，*Alex Sychev*，*Alexey Chernenkov*，*Andrea Spadaccini*，*Andrey Mirtchovski*，*Anthony Magro*，*Babu Sreekanth*，*Ben Bullock*，*Bob Cunningham*，*Brian Fallik*，*Cecil New*，*Damian Gryski*，*Dan Kortschak*，*David Otton*，*Fabian Becker*，*Filip Zaludek*，*Hadi Amiri*，*Haiping Fan*，*Jaap Akkerhuis*，*JC van Winkel*，*Jeroen Bulten*，*Jinpu Hu*，*John Shahid*，*Jonathan Kans*，*Joshua Stein*，*Makoto Inoue*，*Mayuresh Kathe*，*"mem"*，*Michael Stapelberg*，*Olexandr Shalakhin*，*Paulo Pinto*，*Peter Kleiweg*，*Philipp Schmidt*，*Robert Johnson*，*Russel Winder*，*Sonia Keys*，*Stefan Schroeder*，*Thomas Kapplet*，*T.J. Yang*，*"Cobold"*，*"Simoc"*，*"Uriel"*†，*邢星*。

以下诸位提供了细节的改进，例如错误的语法、拼写和其他一些内容：*Alexander Katasonov*, *Daniele Pala*, *Iaroslav Tymchenko*, *Nicolas Kaiser*, *Marco Ynema*.

Miek Gieben



Miek Gieben 从荷兰内梅亨大学取得计算机科学硕士学位。他参与并开发了 DNSSEC 协议——下一代 DNS——DNSSEC protocol 以及如核心认证 [6]。在玩过了 Erlang 后，Go 是他当前迷恋的主要语言。他将所有业余时间用来对 Go 的探索和编码。他是 Go DNS 库的维护者：`https://github.com/miekg/dns`。他的个人博客是 `http://www.miek.nl` 以及 Twitter 帐号 `@miekg`。博文和推多数情况下都是关于 Go 的。

邢星



在你看到本书的中文版的时候，邢星已经从中山大学获得了软件工程硕士学位。他的博客是 `http://mikespook.com` 并且拥有 Twitter 帐号 `@mikespook`。博文和推大部分情况下都是用来扯蛋的，而且会时不时的扯疼……

许可证和版权

B 索引

array
- capacity, 15
- length, 15
- multidimensional, 15

buffered, 90
built-in
- append, 14, 17
- cap, 14
- close, 14
- complex, 14
- copy, 14, 18
- delete, 14
- imag, 14
- len, 14
- make, 14, 54
- new, 14, 54
- panic, 14
- print, 14
- println, 14
- real, 14
- recover, 14

channel, 2, 83
- blocking read, 84
- blocking write, 84
- non-blocking read, 84
- non-blocking write, 84
- unbuffered, 84

channels, 83
closure, 30
complex numbers, 14

deferred list, 30
duck typing, 70

field
- anonymous, 58

fields, 56
function
- as values, 31
- call, 58
- literal, 30
- literals, 31

generic, 74
goroutine, 82
goroutines, 2

interface, 70
- set of methods, 70
- type, 70
- value, 70

io.Reader, 91

keyword
- break, 9, 10
- continue, 11
- default, 12
- defer, 29
- else, 9
- fallthrough, 12
- for, 10
- go, 82
- goto, 10
- if, 8
- import, 45
- iota, 5
- map, 18
  - add elements, 18
  - existence, 18
  - remove elements, 19
- package, 44
- range, 11, 18
  - on maps, 11, 18
  - on slices, 11
- return, 9
- select, 84
- struct, 57
- switch, 12
- type, 57

label, 10
literal
- composite, 15, 56

method call, 58
methods
- inherited, 59

MixedCaps, 46

named return parameters, 26
networking
    Dial, 93
nil, 54

operator
    address-of, 54
    and, 7
    bit wise xor, 7
    bitwise
        and, 7
        clear, 7
        or, 7
    channel, 83
    increment, 54
    not, 7
    or, 7

package
    bufio, 46, 49, 90
    builtin, 14
    bytes, 45
    compress/gzip, 46
    encoding/json, 49
    even, 44
    flag, 49
    fmt, 14, 49
    html/template, 49
    io, 49, 91
    net/http, 49
    os, 49
    os/exec, 50, 92
    reflect, 50, 76
    ring, 46
    sort, 49
    strconv, 49
    sync, 49
    unsafe, 49
parallel assignment, 4, 10
pass-by-value, 26
private, 45
public, 45

receiver, 26
reference types, 15
runes, 11

scope
    local, 27
slice
    capacity, 15
    length, 15
string literal
    interpreted, 6
    raw, 6

toolin
    go
        test, 47
tooling
    go, 3
        build, 3
type assertion, 72
type switch, 71

variables
    _, 4
    assigning, 4
    declaring, 4
    underscore, 4

字段, 58

方法, 26

C Bibliography

[1] Haskell Authors. Haskell. http://www.haskell.org/, 1990.

[2] D. Crockford. The application/json media type for javascript object notation (json). http://www.ietf.org/rfc/rfc4627.txt, 2006.

[3] Brian Kernighan Dennis Ritchie. The C programming language, 1975.

[4] Ericsson Cooperation. Erlang. http://www.erlang.se/, 1986.

[5] Larry Wall et al. Perl. http://perl.org/, 1987.

[6] Kolkman & Gieben. Dnssec operational practices. http://www.ietf.org/rfc/rfc4641.txt, 2006.

[7] Go Authors. Defer, panic, and recover. http://blog.golang.org/2010/08/defer-panic-and-recover.html, 2010.

[8] Go Authors. Effective Go. http://golang.org/doc/effective_go.html, 2010.

[9] Go Authors. Go faq. http://golang.org/doc/go_faq.html, 2010.

[10] Go Authors. Go language specification. http://golang.org/doc/go_spec.html, 2010.

[11] Go Authors. Go package documentation. http://golang.org/doc/pkg/, 2010.

[12] Go Authors. Go tutorial. http://golang.org/doc/go_tutorial.html, 2010.

[13] Go Authors. Go website. http://golang.org/, 2010.

[14] Go Community. Function accepting a slice of interface types. http://groups.google.com/group/golang-nuts/browse_thread/thread/225fad3b5c6d0321, 2010.

[15] James Gosling et al. Java. http://oracle.com/java/, 1995.

[16] LAMP Group at EPFL. Scala. http://www.scala-lang.org/, 2003.

[17] C. A. R. Hoare. Quicksort. http://en.wikipedia.org/wiki/Quicksort, 1960.

[18] C. A. R. Hoare. Communicating sequential processes (csp). http://www.usingcsp.com/cspbook.pdf, 1985.

[19] Rob Pike. The Go programming language, day 2. http://golang.org/doc/{G}oCourseDay2.pdf, 2010.

[20] Rob Pike. The Go programming language, day 3. http://golang.org/doc/{G}oCourseDay3.pdf, 2010.

[21] Bjarne Stroustrup. The C++ programming language, 1983.

[22] Ian Lance Taylor. Go interfaces. http://www.airs.com/blog/archives/277, 2010.

[23] Imran On Tech. Using fizzbuzz to find developers... http://imranontech.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/, 2010.

[24] Wikipedia. Bubble sort. http://en.wikipedia.org/wiki/Bubble_sort, 2010.

[25] Wikipedia. Communicating sequential processes. http://en.wikipedia.org/wiki/Communicating_sequential_processes, 2010.

[26] Wikipedia. Duck typing. http://en.wikipedia.org/wiki/Duck_typing, 2010.

[27] Wikipedia. Iota. http://en.wikipedia.org/wiki/Iota, 2010.

[28] D. Zimmerman. The finger user information protocol. http://www.ietf.org/rfc/rfc1196.txt, 1990.

当前页留空。